Introducción a Python para geociencias

Germán A. Prieto

Facultad de Ciencias
Sede Bogotá



Introducción a Python para geociencias

Introducción a Python para geociencias

Germán A. Prieto



Bogotá, D. C., Colombia, 2024

Primera edición, febrero de 2024

ISBN: 978-958-505-484-4 (digital)

Edición
Daniela Guerrero Acosta
Coordinación de Publicaciones
Facultad de Ciencias
coorpub_fcbog@unal.edu.co

Corrección de estilo
Andrés Manrique Granados

Diseño de la colección
Leonardo Fernández Suárez

Maqueta LaTex
Camilo Cubides

La autoría de las tablas y figuras es del autor, salvo se indique lo contrario.

Catalogación en la publicación Universidad Nacional de Colombia

Prieto Gómez, Germán Andrés, 1979-
Introducción a Python para geociencias / German A. Prieto. – Primera edición. – Bogotá : Universidad Nacional de Colombia. Facultad de Ciencias. Coordinación de Publicaciones Facultad de Ciencias, 2024
1 recurso en línea (xxx, páginas) : ilustraciones (principalmente a color), diagramas, mapas. – (Colección textos)

Incluye referencias bibliográficas e índice
ISBN 978-958-505-484-4 (digital)

1. Ciencias de la tierra – Procesamiento de datos 2. Python (Lenguaje de programación de computadores) 3. Programación orientada a objetos (Computación) 4. Procesamiento electrónico de datos I. Título II. Serie

CDD-23 550.28551 / 2024

Hecho en Bogotá, D. C., Colombia

Contenido

Prefacio

¿Para qué un nuevo libro de Python? ¿Y en español? A pesar de que hay textos disponibles, y muchos ejemplos en la web que sirven para aprender de manera individual, este libro tiene un objetivo específico: es una introducción a la programación con Python para el principiante en computación. Muchos de los libros disponibles se concentran en las capacidades, características y estructura de los códigos de Python, pero no contienen una guía para que el lector aprenda *cómo programar*.

Este libro es a la vez una introducción a Python y una guía de cómo programar. Es un libro adecuado para el geocientífico experto que quiere incursionar en la programación, el profesional recién egresado que busca aplicar métodos computacionales en su trabajo, o el estudiante de pregrado y posgrado que siente deficiencias en aspectos computacionales, y que no encuentra cursos aplicados que puedan ser útiles en el desarrollo de sus estudios y de su futura carrera. Hay pocas referencias con estas características, y en español la disponibilidad es aún menor. Este texto busca cubrir la demanda del público geocientífico en Latinoamérica, y el gran número de estudiantes de pregrado y posgrado en Geología.

El objetivo de este libro es que sea fácilmente entendible, y que le proporcione al lector las herramientas básicas que requiere para poder buscar soluciones a problemas numéricos en geociencias. De este modo, el libro no busca

ser una fuente o referencia para solucionar problemas comunes específicos, sino permitir que el lector pueda, usando Python, generar sus propios algoritmos, y ponerlos en práctica.

Además, a pesar de que cada día se publica software (*módulos* y *paquetes*, muchos con licencias gratuitas) que pueden ser utilizados para solucionar problemas en geociencias, este libro no pretende ser una introducción a una larga lista de software. Sí busca en cambio dar las herramientas para que el lector pueda implementar cualquier algoritmo, y, sobre todo, que tenga la capacidad de **mirar dentro del algoritmo**, cambiar partes del código, adaptarlo a sus necesidades, y evitar así usar software como cajas negras. En otras palabras, que el lector tenga la capacidad de *entender* lo que hace el algoritmo, lo que hay dentro. E incluso, que pueda desarrollar nuevo software. En ese sentido, los únicos paquetes no tradicionales que se utilizan en este libro son paquetes para la generación de mapas.

El enfoque del libro viene con mis prejuicios. Yo aprendí a programar tarde en mi carrera, después de terminar mi pregrado en Geología. Python es un lenguaje orientado a objetos (*object-oriented*), a diferencia de Fortran (F77/F90), el lenguaje que primero aprendí durante mis estudios de posgrado. Por lo tanto, el libro tiene un enfoque más clásico, con poco énfasis en orientación a objetos.

Este libro incluye *scripts* de Python (archivos que contienen código para ser corrido en Python), y *Notebooks* o *Jupyter Notebooks* que replican todos los ejemplos y la mayoría de figuras en el libro. Dichos archivos están disponibles en el material suplementario del libro, y se puede acceder a ellos por medio de github (link).

¿Por qué Python?

El lenguaje de programación Python fue creado alrededor de 1990 por el holandés Guido van Rossum, y su nombre es un tributo al grupo cómico Monty Python. Es un lenguaje dinámico de programación interpretado o de script multiplataforma, con una sintaxis clara y orientada a objetos, que favorece crear código legible y reutilizable. La página web es `www.python.org`.

Prefacio

Python es un lenguaje sencillo de *leer* y aprender, y ha ganado popularidad recientemente. Además, Python es gratuito para todos los usuarios, y se puede instalar en cualquier sistema operativo de uso común en los computadores personales y de oficina.

En tanto, los lenguajes compilados (C, C++, Fortran) son mucho más rápidos, y son la base de algoritmos usados en supercomputación. Es difícil superar estos lenguajes en velocidad y capacidad. Sin embargo, dichos lenguajes son difíciles de aprender, con ellos se debe compilar el código antes de ver su resultado, y no pueden generar gráficas de manera sencilla (hay paquetes para ello, claro). En suma, son particularmente difíciles de aprender para un no-programador.

Matlab también es un lenguaje interpretado, y tiene algunas características que lo hacen muy interesante. Cuenta con una colección de librerías muy grande para múltiples aplicaciones, desarrolladas en lenguajes compilados, que le permiten ser rápido. Su ambiente de trabajo es amigable, y tiene un soporte comercial. Su gran desventaja es que no es un programa gratuito. Otros algoritmos como Julia, R, Octave, etc., son en su mayoría gratuitos, y algunos tienen librerías muy eficientes. Sin embargo, el aporte actual de nuevos paquetes en Python se ha acelerado, lo cual le da alguna ventaja.

Como parte de la instalación básica de Python, se incluyen paquetes para procesamiento y análisis de datos como *Pandas*, *SciPy* y *NumPy*, que incluyen métodos estadísticos, transformadas de Fourier, generación de figuras, etc. Al mismo tiempo, hay una gran cantidad de paquetes, también gratuitos, en Python para geociencias [43], sismología [3, 17], geofísica [6], generación de mapas [41, 24], ciencias atmosféricas [4], análisis espectral [26], etc.

Con un manejo adecuado de tales herramientas, el geocientífico puede obtener, descargar, y simular datos, poder manipularlos y hacer un procesamiento adecuado, y visualizar los resultados con figuras de alta calidad, listas para publicación.

Capítulo 1

Software e instalación

python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt
n3.x python3.x python3.x python3.x python3.x python3.x python3.x
python3.x python3.x python3.x python3.x python3.x python3.x pyt

Python puede ser instalado en cualquier sistema operativo, por ejemplo, por medio de una de las plataformas de distribución mas populares: Anaconda. Anaconda es una distribución libre y abierta de Python y R, y funciona como un administrador de paquetes, haciendo muy sencillo organizar, instalar y actualizar software. Anaconda contiene las librerías más comunes para computación científica y generación de gráficas de alta calidad, y permite la administración de *virtual environments* (entornos virtuales), e incluye Jupyter Notebooks. De forma alternativa, Google ofrece Colaboratory o Colab (colab.research.google.com), con el cual se pueden ejecutar *notebooks* en la nube con servidores de gran potencia.

1. Instalación de Python con Anaconda

La instalación de Python a través de Anaconda es muy sencilla en cualquier sistema operativo. La página de Anaconda es `anaconda.org`, o el link para descargar es `https://www.anaconda.com/download/`. La página web detecta el sistema operativo y provee el enlace para descargar el programa en su última versión estable. Se recomienda instalar la versión para Python 3.X, no la versión 2.7.

Después de descargar el programa de instalación (para cualquier sistema operativo), se siguen los comandos para instalar la versión de Anaconda.

2. Cómo usar Python

El programa Python (como programa, no como lenguaje) tiene un ambiente interactivo que permite ejecutar instrucciones del lenguaje Python de forma directa. Si tiene un sistema operativo Mac o Linux, en la terminal, o si tiene un Windows en el *Anaconda Prompt* basta con digitar el comando en la linea de comandos:

```
> python
```

En tal caso, se desplegará algo similar a:

```
Python 3.7.6 | packaged by conda-forge | ...
Type "help", "copyright", "credits" or "license" ...
>>>
```

El programa ofrece un prompt (>>), que espera instrucciones del usuario, de modo similar a lo que sucede con Matlab. Las instrucciones son interpretadas y ejecutadas de manera inmediata, así:

```
>>> 2 + 3
5
>>>
```

Esta forma de interactuar con Python es útil, ya que permite obtener respuestas inmediatas, y se puede determinar si el comando introducido es correcto. Sin embargo, tiene la desventaja de que el código y la secuencia de comandos que se hayan utilizado no se guardan, y, por lo tanto, al siguiente día se debe repetir toda la operación. Para salir digite *exit()*.

Si el sistema operativo es Windows, esto es diferente, ya que Windows no está basado en un sistema tipo-unix, como Mac o Linux. La instalación de Anaconda incluye una terminal con el nombre de *Anaconda prompt*, en la que se pueden seguir los pasos anteriores.

2.1. *Scripts*

Una segunda opción para interactuar con Python es a través de archivos de texto o *scripts*, los cuales pueden tener una serie de comandos que Python va a leer, interpretar y ejecutar en el orden dado por el *script*. Este archivo puede ser reutilizado; sin embargo, la retroalimentación de Python solo se da cuando se ejecute el *script*.

Estos *scripts* deben ser archivos de texto plano, cuyo nombre debe terminar con *.py*. No es buena idea nombrar los archivos con espacios, es preferible usar conectores, por ejemplo *un_programa.py* en vez de *un programa.py*. Para editar

estos archivos, se puede usar editores de texto plano como *Vi, Emacs, Gedit* o plataformas de desarrollo como *Spyder*, muchos incluidos en Anaconda.

Para ejecutar el programa en la terminal, utilizar:

```
> python un_programa.py
```

Python ejecutará todos los comandos que haya en *un_programa.py* en el orden en que estén escritos.

2.2. Jupyter Notebooks y Colab

Para el programador experto (o de vieja guardia, como yo), la interacción con cualquier lenguaje de programación se da mediante *scripts* o código fuente, que luego es ejecutado en Python (o compilado por Fortran, C, etc.). Sin embargo, hoy los códigos de Python (R, y otros lenguajes) pueden ser ejecutados, visualizados y compartidos con otras personas por medio de Jupyter Notebooks o Colab. La figura 1.1 muestra un ejemplo de un *Jupyter notebook* en el que se pueden escribir código y comentarios con un formato amigable. Además, se puede ejecutar el código por partes, e incluso desplegar figuras dentro del mismo *notebook*. En recientes trabajos, se incluyen *notebooks* para aumentar la difusión y el aprendizaje de Python en la comunidad científica [21, 1, 7].



Figura 1.1. Ejemplo de un *Jupyter notebook*.

Jupyter es una aplicación web de código abierto (open-source) que permite crear y compartir código Python, al igual que visualizar figuras dentro del programa, e incluir notas de texto, comentarios, etc. Los *notebooks* son fáciles de guardar y compartir, y son hoy en día aceptados y usados por la comunidad científica global [34]. La ventaja de Jupyter Notebooks es que se usa a través de un navegador de internet (Explorer, Chrome, Firefox, etc.). Esto permite que, sin importar el sistema operativo (Windows, Mac, Linux), los programas de Python funcionen de igual manera en cualquiera de ellos.

Tanto en Linux como en Mac, la correcta instalación de Anaconda permite llamar a Python desde la terminal. Para abrir un *notebook*, basta con digitar:

```
> jupyter notebook
```

O, en su defecto (macs):

```
> jupyter-notebook
```

En Windows, se puede acceder a Jupyter con el *cmd* o a través del *Anaconda prompt*, digitando *jupyter notebook*.

Colaboratory o Colab es un servicio de Jupyter Notebook en la nube, que permite escribir, organizar y **ejecutar** código de Python desde un navegador. La ventaja que ofrece es el uso de recursos computacionales poderosos de manera *gratuita* para aquellos que tienen cuentas con Google. En principio, funciona de igual manera a un *Jupyter notebook*, y es una alternativa que no requiere la instalación de programas. La figura 1.2 muestra el mismo *notebook* en Colab.

3. Instalación de paquetes

Aunque el objetivo de la *Introducción a Python para geociencias* es aprender a programar en Python, y no es una introducción a paquetes específicos de Python para geociencias, sí es necesario el uso de paquetes adicionales de Python, dentro de los que se incluyen los siguientes:

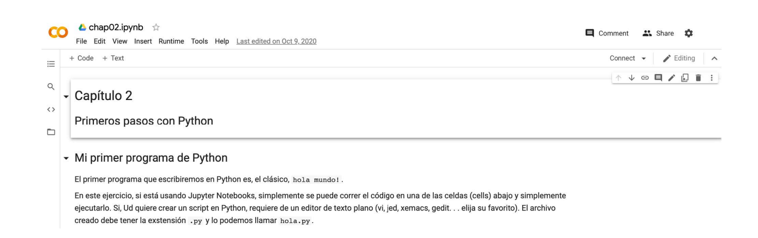


Figura 1.2. Ejemplo de un *Colab notebook* (igual al de la figura 1.1).

- NumPy
- SciPy
- Pandas
- Jupyter
- Matplotlib
- GMT y PyGMT
- Cartopy

Su instalación se puede hacer a través del *Anaconda prompt*. Para mayor facilidad, se incluye un archivo *geopython.yml* que permite la creación de un *environment* de Python que contiene los paquetes necesarios para todos los ejemplos de este libro. Se recomienda crear estos *ambientes* donde se instalan los paquetes que se van a utilizar, y que no afectan el comportamiento de otros ambientes, ni de la instalación base de Python. En tanto, para correr la instalación del *environment*, digitar:

```
conda env create -f geopython.yml
```

En tal caso, se asume que se parte por estar situado en el fichero que contiene el archivo *geopython.yml*, o se debe indicar la ubicación exacta del archivo. A su vez, para activar y usar el *environment*, se usa:

```
conda activate geopython
```

O bien, en algunos casos:

```
source activate geopython
```

Por último, para desactivar el ambiente, se usa:

```
conda deactivate
```

geopython.yml

```
name: geopython
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.7
  - numpy
  - scipy
  - pandas
  - ipython
  - jupyterlab
  - jupyter
  - pip
  - matplotlib
  - netcdf4
  - packaging
  - gmt
  - pygmt
  - obspy=1.1
  - pyasdf
  - cartopy
```

Capítulo 2

Primeros pasos en Python

for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x
range for i in range for i in range for i in range for i in range for i in range for i in
y = x y = x y = x y = x y = x y = x
for i in range for i in range for i in range for i in range for i in range for i in range
y = x y = x y = x y = x y = x y = x

1. Primer programa

El primer programa que escribiremos en Python es, el clásico, *¡hola mundo!*. Para crear este *script* en Python, requerimos de un editor de texto plano (Vi, Jed, Emacs, Gedit. . . elija su favorito). El archivo creado debe tener la extensión *.py* y lo llamamos *hola.py*. Si está corriendo Jupyter Notebooks, lo que se muestra a continuación debe estar dentro de una celda de código. En ambos casos, el código contiene:

hola.py

```
print ("¡Hola Mundo!")
```

Una segunda versión de este programa, permite adicionar comentarios explicativos, haciéndola más informativa:

hola1.py

```
# Mi primer programa de Python

print ("¡Hola Mundo!")
```

Para ejecutar el programa desde la terminal, en la línea de comandos se digita:

```
> python hola.py
```

Como resultado, se obtiene:

```
¡Hola Mundo!
```

Cabe señalar que la ejecución del programa se hace en la línea de comandos, y no abriendo Python. Sin embargo, si se tiene abierto Python, se puede ejecutar el programa dentro de Python. Si se está ejecutando el código por medio de Jupyter Notebooks, solo necesita ejecutar la celda con el código, y el resultado saldrá desplegado abajo del bloque de código.

1.1. Explicación del primer programa

La primera línea del código *hola1.py* es un comentario. Cualquier texto después de un símbolo # es un comentario, y Python no lo tiene en cuenta. Además, al terminar la línea, termina el comentario. No hay necesidad de acabar el comentario con otro símbolo (aunque en otros lenguajes eso sí es necesario). También es posible hacer comentarios dentro de una línea. Por ejemplo:

```
print ("¡Hola Mundo!")    # imprimir texto
```

Es válido poner comentarios a la derecha de cualquier comando.

La segunda línea es una línea en blanco. Python no tiene en cuenta estas líneas, y se pueden poner tantas líneas en blanco como se quiera, ya que serán ignoradas por Python. El uso de líneas en blanco permite hacer el código más fácil de leer, y también permite escribir el código en bloques pequeños.

La siguiente línea sí es interpretada por el programa. El comando *print* ordena que en la terminal se muestre el texto que está dentro del paréntesis. Para el caso de conjuntos de caracteres, estos deben ir entre comillas o comillas dobles.

Otra alternativa de comentario (multilíneas) es usar triple comillas. Este tipo de comentario se conoce como un *docstring*, y permite documentar un segmento específico del código. El *docstring* debe cerrarse con triple comillas.

hola2.py

```
"""
esto es un docstring, es documentación.
No es ejecutado por Python
"""

print("¡Hola Mundo!")    # imprimir texto
```

1.2. Alternativas para el primer programa

Python permite variaciones al programa anterior que también funcionan:

hola3.py

```
# Otras alternativas para el programa

# Asignar variable x
x = "¡Hola Mundo!"

# Imprimir
print(x)
```

En este caso, *x* es una variable de caracteres. Otra alternativa también permite realizar operaciones con variables:

hola4.py

```
# Otras alternativas para el programa
"""
Asignar tres variables (x,y,z), pegarlas para
el mismo resultado.
"""


x = "¡Hola"
y = "Mundo"
z = "!"

print(x+" "+y+z)
```

Cabe anotar que en este último ejemplo, realizamos una operación para *pegar* o *concatenar* variables de caracteres. Con el fin de separar las palabras entre las variables x y y, se pone un espacio. El resultado de estos dos programas (*hola3.py* y *hola4.py*) es idéntico al primer ejemplo, *hola.py*.

2. Multiplicar dos números enteros

Ahora, nos disponemos a realizar operaciones matemáticas en Python. Por ejemplo, un programa simple para multiplicar dos enteros (2 y 3).

multint.py

```
# Código para multiplicar dos números enteros

a = 2
b = 3
c = a*b

print(a,"x",b," = ",c)
```

El programa usa tres variables, las letras *a*, *b* y *c*. Las variables pueden tener nombres largos, pero no pueden tener espacios. Si quiere unir palabras use raya al piso, _. El primer caracter de una variable debe ser una letra, el resto puede ser una combinación de letras, números y _. No se recomienda usar puntos o símbolos de menos (-). A diferencia de Fortran o C, Python define de manera automática si un número es entero o real, aunque si el usuario así lo prefiere, puede definir la variable como número entero:

```
a = int(2)
b = int(3)
```

Cuando esto sucede, Python automaticamente determina que *c* será un número entero. El resultado del programa anterior da como producto:

```
> python multint.py
2 x 3  =  6
```

El símbolo * indica multiplicación como casi en la mayoría de lenguajes de programación. Adición es +, resta es - y división es /. En Python, para elevar un número a a la potencia b, se escribe *a****b*. Las dos variables pueden ser enteros, reales y complejos.

Otro ejemplo, para elevar un número real *a* la potencia *b*, es el siguiente:

$$c = a^b$$

real_elev.py

```
a = 2.01
b = 3.2
c = a**b

print(a, "elevado a la ",b," = ", c)
```

El anterior caso da como resultado:

```
2.01 elevado a la  3.2  =  9.337430530829119
```

3. *For loops*

Cualquier lenguaje de programación debe permitir realizar una serie de operaciones de manera repetida. Esto significa permitir realizar un *loop* o *bucle* para una serie de valores de una variable. En C o Matlab, esto se lleva a cabo con un *for loop*, en Fortran se hace con un *do loop*. En Python se usa el *for*.

A continuación, un ejemplo simple para generar la tabla de multiplicar del 7 ($7 \times 0, 7 \times 1, 7 \times 2$, etc.):

mult_table.py

```
x = 7    # la tabla del 7

for y in range(10):       # loop 0, 1, 2, ..., 9
    z = x*y
    print(x,'x',y,' = ',z)
print('Acabó el loop')
```

```
7 x 0  =  0
7 x 1  =  7
...
7 x 8  =  56
7 x 9  =  63
```

El *for loop* es un comando que ejecuta una serie de comandos (que deben estar indentados). El número de veces o la forma como los ejecuta depende del código, a saber:

```
for y in range(10):
```

Acá, la variable *y* toma valores en un rango de 0 hasta 9. Es decir, repite la operación 10 veces, empezando en 0, hasta llegar al 9. ¿Por qué Python no incluye el 10? El comando *range* genera 10 números, pero Python (como C) empieza el conteo con el cero; es decir, 0, 1, 2, ..., 9, para un total de 10 números. Como se puede ver, en el primer *loop*, $y = 0$; en el segundo, $y = 1$, y así sucesivamente. Una segunda versión del mismo código usando *numpy*, es:

mult_table2.py

```
import numpy as np

x = 7         # la tabla del 7
i = np.arange(10)

for y in i:       # loop 0, 1, 2, ..., 9
    z = x*y
    print(x,'x',y,' = ',z)
```

El resultado de esta segunda versión del código es similar al anterior, pero tiene variaciones. Python tiene una larga lista de librerías (conocidas como *modules* o *packages*) para realizar operaciones de todo tipo. Para poder utilizarlas en un programa, es necesario cargarlas. Esto se realiza con el comando *import*.

En este caso, importamos el paquete NumPy, a saber:

```
import numpy as np
```

NumPy [14] tiene múltiples funciones para realizar operaciones en variables, vectores, etc. Más adelante, veremos el uso de paquetes y módulos, incluyendo *NumPy*.

Para usar una función dentro de NumPy, se debe cargar el paquete *numpy* y llamar la función *arange*. Para que esta operación no sea muy larga, le ponemos un nombre más corto, *np*. Esto permite usar la función *arange* dentro de NumPy, y generar un vector:

```
i = np.arange(10)
```

Este comando genera un arreglo (o vector), con valores desde 0 hasta 9. Es algo muy similar a lo que hace *range(10)*, pero lo pone en el arreglo *i*.

3.0.1. Sobre indentación y *loops*

La indentación es importante y debe tener en cuenta:

- No mezcle espacios y *tabs*.
- Python busca un número de espacios exacto en la indentación.
- Aunque no se vea, un *tab* no es equivalente a ocho espacios para Python.
- Muchos *notebooks* hacen la indentación por el usuario, pero debe procurar no confiarse.

Asimismo, deberá tener precauciones al abrir y cerrar *loops*:

- La línea con el comando *for* debe terminar con dos puntos (:).
- Todo lo que se quiere dentro del *loop* debe estar indentado.
- Para terminar el *loop*, retire la indentación.
- A diferencia de otros lenguajes, Python no tiene un *end* para el *loop*.

3.1. Una tabla trigonométrica con *for loops*

A continuación, se muestra un ejemplo de código para generar una tabla de funciones trigonométricas (seno, coseno y tangente) para diferentes ángulos (*ang* = *0.0, 1.0, 2.0, etc.*).

trigtable.py

```
import math
import numpy as np

i = np.arange(90)

# Haga un loop
for ang in i:
    theta  = ang/180.*math.pi  # transformar ángulo
    ctheta = math.cos(theta)
    stheta = math.sin(theta)
    ttheta = math.tan(theta)
    print(ang, ctheta, stheta, ttheta)
```

El resultado es:

```
0 1.0 0.0 0.0
1 0.99984769515... 0.017452406437... 0.0174550649282...
2 0.99939082701... 0.034899496702... 0.034920769491...
...
88 0.034899496702... 0.99939082701... 28.6362532829...
89 0.01745240643... 0.99984769515... 57.2899616307...
```

Las funciones trigonométricas no forman parte de las funciones estándar de Python, por lo tanto se importa el módulo *math* con el siguiente comando:

```
import math
```

El módulo contiene el valor de π en la variable *math.pi*. Además, tiene funciones para calcular coseno (*math.cos*), seno (*math.sin*) y tangente (*math.tan*), y mucho más (ver ejemplos mas adelante).

Tal como lo hacen C, Fortran o Matlab, el cálculo trigonométrico se hace en radianes (no grados) como argumento en las funciones trigonométricas. Por ello, es necesario convertir los ángulos en grados (α), a radianes θ.

$$\theta = \alpha \frac{\pi}{180}$$

Aunque el anterior resultado es correcto, la tabla que se muestra no es fácil de leer. De tal modo, para mejorar el formato de salida del resultado, se usa el siguiente procedimiento:

trigtable2.py

```
import math
import numpy as np

i = np.arange(90)

for ang in i :
    theta = float(ang/180.*math.pi)
    ctheta = math.cos(theta)
    stheta = math.sin(theta)
    ttheta = math.tan(theta)

    # Desplegar resultados más amables
    print ("%5.1f, %7.4f, %7.4f, %8.4f"
           % (ang,ctheta,stheta,ttheta))
```

```
  0.0,  1.0000,  0.0000,   0.0000
  1.0,  0.9998,  0.0175,   0.0175
  2.0,  0.9994,  0.0349,   0.0349
...
```

```
 88.0,  0.0349,  0.9994,  28.6363
 89.0,  0.0175,  0.9998,  57.2900
```

El código hace la misma operación para el ángulo *ang*:

```
for ang in i :
    theta = float(ang/180.*math.pi)
    ctheta = math.cos(theta)
   ...
```

Esta toma sucesivamente los valores del arreglo *i*, es decir en cada iteración *0, 1, 2, ..., 88, 89*. A partir de ese valor, se define el ángulo *theta* en radianes, usando la función *float*. Es decir, *theta* asume valores sucesivos de *0.0, 1.0, 2.0, ..., 88.0, 89.0* en cada iteración, convertidos a radianes. Recomiendo siempre realizar las iteraciones sobre un vector de números enteros (*i*), ya que errores de *redondeo* pueden evitar que se llegue exactamente hasta *89.0*.

Dentro de cada *for loop*, se calculan los cosenos, senos y tangentes del ángulo *theta*. Por último, en cada *loop* se muestran los resultados:

```
print ("%5.1f, %7.4f, %7.4f, %8.4f"
          % (theta,ctheta,stheta,ttheta))
```

De tal modo, el comando *print* tiene un formato especificado por el usuario. En este caso, *%5.1f* ordena que la variable *theta* se imprima como un número real con 5 espacios en total y 1 dígito a la derecha del punto decimal. De manera similar, el *7.4f* ordena que *ctheta* tenga 7 espacios y 4 números a la derecha del decimal. Los números están justificados a la derecha. Python permite la continuidad del comando dentro del paréntesis. Aquello significa que el comando en el bloque de código arriba, es un solo comando.

3.2. Posibles variaciones

El código expuesto a continuación sugiere dos posibles variaciones que pueden ser útiles.

trigtable3.py

```
from math import *
import numpy as np

# Defina el formato acá
fmt = "%5.1f, %7.4f, %7.4f, %8.4f"

i = np.arange(90)

for ang in i :

    theta  = float(ang/180.*pi)
    ctheta = cos(theta)
    stheta = sin(theta)
    ttheta = tan(theta)

    # Imprima al resultado
    print(fmt
        %(ang, ctheta, stheta, ttheta))
```

En ellas, se importan todas las funciones dentro del módulo *math*, de tal forma que no hay necesidad de llamarlos con *math.cos*, sino directamente con *cos*. Además, se puede definir una variable *fmt*, que tiene los caracteres que definen el formato de salida. Esto es útil cuando se van a utilizar comandos *print* en múltiples casos dentro de un programa. El módulo *math* tiene guardado el valor de π en la variable *pi*.

4. Funciones del módulo *math*

Acá se muestran otras (no todas las) funciones que se encuentran en el módulo, y que pueden ser útiles para el análisis de datos.

```
ceil(x).        entero próximo mayor o igual a x
fabs(x)         valor absoluto de x
floor(x)        entero próximomenor o igual a x
isinf(x).       Verifica si x es infinito (+ o -)
isnan(x)        Verifica si x is un NaN (no un número)
exp(x)          exponencial de x
log10(x)        logaritmo base 10
log(x)          logaritmo natural de x
sqrt(x)         raíz cuadrada de x
acos(x)         arco coseno
asin(x)         arcoseno
atan(x)         arco tangente
atan2(y,x)      arco tangente de y/x en cuadrantes
cos(x)          coseno
cosh(x)         coseno hiporbólico
sin(x)          seno
sinh(x)         seno hiporbólico
tan(x)          tangente
tanh(x)         tangente hiporbólica
factorial(n)    factorial, producto de los enteros hasta n
degrees(x)      Convierte ángulo de radianes a grados
radians(x)      Convierte ángulo de grados a radiantes
```

En tanto, las siguientes son las constantes dentro del módulo:

```
pi     valor de pi=3.14159265...
e      valor de la constante e=2.718281...
tau    valor de 2*pi, tau=6.2831...
```

5. Sobre formatos

Python permite desplegar en la terminal los números y caracteres en diferentes formatos. Acá algunos ejemplos útiles:

```
%5i   =  entero, 5 espacios, justificado a la derecha

%5.4i =  igual, pero con ceros a la izquierda (88 es 0088)

%8.3f =  real, 8 espacios, 3 a la derecha del punto decimal

%12.4e = real con exponente, 4 valores a la derecha del
         decimal p.e., b-0.2342E+02 donde "b" es un espacio
         en blanco (sirve para escribir valores grandes y
         pequeños, o cuando no se sabe qué tamaño tendrán
         los valores)

%8s  =  lista de caracteres, 8 espacios, justificado a
        la derecha. Si la lista de  caracteres es mayor
        a 8, se imprimen todas.

%.8s =  igual, pero solo los primeros 8 caracteres se
        imprimen.
```

Problemas

1. Escriba un programa de Python que imprima una frase célebre.
2. Escriba un programa en Python que imprima en la pantalla una tabla con:

```
x    sinh(x)    cosh(x)
```

A saber, las funciones hiperbólicas del seno y coseno, para valores de *x* entre *0.0* a *6.0*, con incrementos de *0.5*. Utilice un formato para el resultado que sea presentable. ¿Será necesario convertir *x* a radianes?

3. Modifique el programa inicial, para mostrar en la pantalla dos números enteros, y el resultado de las cuatro operaciones, *+, -, *, /*. Use un formato de impresión adecuado para cada caso.

4. Genere una tabla de multiplicar para los números del 1 al 9, es decir, *1x1, 1x2, ..., 1x10*, la siguiente línea, *2x1, 2x2, 2x10*, y así hasta el 9.

5. Genere una tabla con los siguientes valores:

$$x \quad \log 10(x) \quad \log(x) \quad \sqrt(x)$$

Haga tantos *loops* como sea posible, empezando con $i = 0, x = 1.01$, para el siguiente *loop*:

$$\begin{aligned} i &= 0, & x_i &= 1.01 \\ i &= 1, & x_i &= x_{i-1}(i+1) \\ i &= 2, & \cdots \end{aligned}$$

En cada *loop*, calcule los logaritmos y la raíz cuadrada. Eventualmente, el resultado no va a caber en el computador (los números serán muy grandes). En su equipo, ¿cuándo obtiene un resultado *inf* o similar? De este modo se muestra el límite de precisión del computador.

Los primeros valores de x deben ser *1.01, 2.02, 6.06, 24.24, ...*

6. Asumiendo que la velocidad absoluta de una placa tectónica es de 55 mm/a, calcule a qué distancia de la dorsal se encuentra una roca de 1 Ma, de 8 Ma y de 60 Ma.

7. Una placa tectónica tiene una velocidad horizontal de 60 mm/a, y subduce con un ángulo de 60 grados por debajo de otra placa. ¿A qué profundidad estaría la placa después de 1 Ma, de 5 Ma y de 40 Ma de comenzar a subducir? ¿Cuánto tardaría la placa en llegar al límite manto superior - manto inferior (660 km)?

Capítulo 3

Interacción con Python

```
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
put('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     inp
while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):
    input('Digi     input('Digi     input('Digi     input('Digi     input('Digi     input('Digi
e (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while (a!=0):  while
```

Hasta el momento, todos los ejemplos han sido de programas que no requieren o solicitan información del usuario. Aquello impone la limitación de que el programa es estático, y siempre se va a comportar de la misma manera. Por ejemplo, el programa *multint.py* (sección 2) siempre dará como resultado 6, y si queremos hacer una multiplicación diferente, es necesario escribir un nuevo programa.

1. Entrada con el teclado

Para ampliar la capacidad de los programas de Python, estos pueden solicitar información del usuario a través del teclado. Para esto, se le debe indicar al usuario qué información dar, como en el siguiente ejemplo:

usuario_info.py

```
# Código que solicita información del usuario
# y reporta los resultados

name   = input("Cuál es su nombre? ")
txt    = input('Cuál es su estatura (m)? ')
alt    = float(txt)
txt    = input('Cuánto pesa Ud. (kg)? ')
weigh  = float(txt)

print ('%s, Ud. mide %4.2f m y pesa %5.1f kg'
       % (name,alt,weigh))
```

De tal modo, en la terminal da como resultado lo siguiente:

```
Cuál es su nombre? Germán
Cuál es su estatura (m)? 1.82
Cuánto pesa Ud. (kg)? 87
Germán, Ud. mide 1.82 m y pesa  87.0 kg
```

Como se puede ver, el comando *input* es el encargado de solicitarle al usuario que digite una respuesta con el teclado. El programa no continuará a menos que el usuario oprima Enter.

De esta forma, para los valores solicitados de estatura o peso, con el siguiente comando se genera una variable *txt* con caracteres (una variable tipo *string*):

```
txt    = input('Cuál es su estatura (m)? ')
alt    = float(txt)
```

Posteriormente, la función *float(txt)* convierte la variable *txt* en una variable real o *float*. Cabe anotar que si el usuario responde con letras o símbolos, el programa generará un error; es decir, se espera un número como respuesta.

Como se ve en el programa anterior, solicitar información del usuario es bastante sencillo en Python. El siguiente programa muestra una variación al programa *multint.py*, de tal forma que los números a ser multiplicados sean definidos por el usuario.

usuariomult.py

```
# Código para multiplicar dos números enteros,
# definidos por el usuario.

a = int(input("Digite un primer número entero:  "))
b = int(input("Digite un segundo número entero: "))

c = a*b
print(a,"x",b," = ",c)
```

Este tiene como resultado lo siguiente:

```
Digite un primer número entero:  3
Digite un segundo número entero: 5
3 x 5  =  15
```

Este código tiene dos llamadas para pedirle al usuario que digite los números a multiplicar. También es importante notar lo que sucede al poner el siguiente código:

```
a = int(input("Digite un primer número entero:  "))
```

En tal caso, se le pide al usuario un número entero, pero si el usuario digita un número real (2.2) o caracteres diferentes a números, el programa mostrará un error:

```
Digite un primer número entero:  2.2
-----------
ValueError
Traceback (most recent call last)
<ipython-input-8-7860f1ff099c> in <module>
      3 # definidos por el usuario.
      4
----> 5 a = int(input("Digite un primer ...
      6 b = int(input("Digite un segundo ...
      7

ValueError: invalid literal for int() with
    base 10: '2.2'
```

En esa situación, la respuesta y explicación del error es bastante clara, y muestra el problema que se dio.

Con todo, el anterior programa tiene una desventaja: al usuario se le solicita la pareja de números uno a uno. Abajo se muestra entonces una versión más corta, en la que se le solicitan al usuario los dos números con una sola llamada. El usuario debe digitar los números con un espacio entre ellos, no con coma (,), ni otro separador. Tampoco puede digitar Enter entre los números, pues eso generará un error.

usuariomult2.py

```
# Código para multiplicar dos números enteros,
# definidos por el usuario.
# Una sola solicitud para los dos números.
#

intxt = input("Digite dos números enteros: ")
a,b = intxt.split()
a = int(a)
b = int(b)

c = a*b

print(a,"x",b," = ",c)
```

En efecto, el programa generará el siguiente resultado:

```
Digite dos números enteros: 2 3
2 x 3  =  6
```

El ejemplo anterior muestra que el comando *input* permite digitar una serie de caracteres. Si esos caracteres están separados por espacios, se pueden asignar a múltiples variables con el comando *intxt.split()*. Sin embargo, si el usuario digita tres números (en vez de dos), se genera un error. ¿Puede el lector pensar en alguna alternativa para evitar este error?

2. *For* y *while loops*, condicionales *if*

Los programas anteriores siguen mostrando limitaciones. Aunque el usuario puede hacer la multiplicación, tiene que volver a correr el programa si quiere cambiar los números. ¿Qué hacer si se quiere hacer múltiples operaciones con diferentes pares de valores?

¿Cómo hacer un programa que le solicite reiteradamente al usuario dos números hasta que este quiera detenerse?

usuariomult3.py

```
# Código para multiplicar dos números enteros,
# definidos por el usuario.
# Operación se repite hasta que el usuario lo determine.
#

for i in range(1000):
    intxt = input('Digite dos enteros (0 0 para) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
    if (a==0 and b==0):
        break
    c = a*b
    print(a,"x",b," = ",c)
```

```
Digite dos números enteros (0 0 para) 2 5
2 x 5  =  10
Digite dos números enteros (0 0 para) 7 9
7 x 9  =  63
Digite dos números enteros (0 0 para) 1 2
1 x 2  =  2
Digite dos números enteros (0 0 para) 0 0
```

Este código puede no ser el mas óptimo. Python no permite hacer *loops* de manera indefinida (un infinito número de veces), así que debemos indicar que lo haga en un rango definido (hasta 1000 veces en nuestro ejemplo). Sin embargo, no sería muy práctico que el usuario tuviera que hacer la operación mil veces para que el programa termine. Por esa razón, existen los condicionales, de modo que al cumplirse cierta condición (que ambos números sean 0

en nuestro ejemplo), el programa hace un *break* que le ordena salir del *for loop*. Cabe destacar que el código dentro del *loop* está indentado.

El programa le seguirá pidiendo al usuario los dos números, e imprimiendo el resultado, hasta que el usuario lo detenga. Para detener el programa, los dos números deben ser cero (0), como es evaluado con el siguiente condicional:

```
if (a==0 and b==0):
```

Así, el resultado de correr el programa se muestra arriba, y es terminado en la última linea, como se muestra a continuación:

```
Digite dos números enteros (0 0 para) 0 0
```

Operadores de relación en varios lenguajes

F77	F90	C	MATLAB	Python	significado
.eq.	==	==	==	==	es igual
.ne.	/=	!=	~=	!=	no es igual
.lt.	<	<	<	<	menor a
.le.	<=	<=	<=	<=	menor o igual a
.gt.	>	>	>	>	mayor a
.ge.	>=	>=	>=	>=	mayor o igual a
.and.	.and.	&&	&	and	y
.or.	.or.	||	|	or	o

En Python, el *for loop* procesa cada elemento dentro de una secuencia, así que puede ser usado en cualquier secuencia de datos de diferente tipo (*arreglos, strings, listas, tuples*, etc.). A la variable del *loop* (*i*, en este ejemplo) se le asigna en cada iteración el valor correspondiente, y el cuerpo (indentado) dentro del *loop* es ejecutado. La forma general de un *for loop* en Python es la siguiente:

```
for VARIABLE_LOOP in SECUENCIA:
    Comandos a ejecutar
```

Debe anotarse que los comandos tienen un encabezado (*header*) que termina con dos puntos (:), y un cuerpo con una serie de comandos indentados. La indentación puede ser de uno, dos, o cualquier número de espacios, pero estos siempre deben tener la misma longitud.

Asimismo, en lenguajes modernos, se tiene la opción de utilizar un *while loop*, en lugar de un *for loop*. El *while* es un comando compuesto que tiene un encabezado y un cuerpo, con el siguiente formato general:

```
while expresión_lógica:
    Comandos a ejecutar
```

A su vez, el *while loop* se ejecuta de manera continua, siempre y cuando la expresión lógica (booleana, de falso o verdadero) sea cierta. El anterior programa se puede escribir utilizando un *while*:

usuariomult4.py

```
# Operación se repite con while loop.

a=None
b=None
while (a!=0 or b!=0):
    intxt = input('Digite dos enteros (0 0 termina) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
```

```
    c     = a*b
    print(a,"x",b," = ",c)
```

```
Digite dos enteros (0 0 termina) 2 5
2 x 5  =  10
Digite dos enteros (0 0 termina) 7 9
7 x 9  =  63
Digite dos enteros (0 0 termina) 1 2
1 x 2  =  2
Digite dos enteros (0 0 termina) 0 0
0 x 0  =  0
```

Cabe señalar que, para iniciar el *while loop*, es necesario tener definidas las variables *a* y *b*, para que el programa pueda realizar la verificación del condicional la primera vez.

2.1. Escoger entre *for* y *while loops*

Hay entonces dos tipos de *loop*. Para el programador clásico, el *for loop* parece más sencillo y lógico. Pero, ¿cuál escoger?

- Si el programador sabe •e antemano el número de iteraciones que debe hacer, o va a hacer un *loop* sobre una lista o un arreglo en el cual el número total de elementos es conocido, el *for loop* es su mejor opción.

- Si debe hacer una iteración de un cálculo hasta que una condición se cumpla, y no se puede saber cuándo esa condición se va a cumplir, el *while loop* es su mejor opción.

El primer caso se conoce como una *iteración definida*, mientras que el segundo caso es conocido como *iteración indefinida*, ya que no sabemos de antemano cuántas iteraciones son requeridas.

3. Múltiples condicionales *if, elif, else*

En el anterior ejemplo, una operación es llevada a cabo si una condición se cumple. Una versión más versátil permite evaluar múltiples condiciones con la siguiente estructura:

```
if (expresión_lógica):
   (bloque de código)
elif (expresión_lógica):
   (bloque de código)
elif (expresión_lógica):
   (bloque de código)
...
else:
   (bloque de código)
```

Cada bloque de código puede tener tantas líneas y comandos como se requiera. Además, puede haber tantos bloques de condiciones *elif* (*else if*) como se requiera, y máximo un *else*. Cuando una condición se cumple, ese bloque de código se ejecuta, sin seguir mirando las otras condiciones posteriores. Es decir, el orden de los condicionales importa (Python no revisa lo que sigue). El último *else* será ejecutado si ninguna condición anterior es verdadera.

El siguiente programa muestra un ejemplo en el que el usuario debe adivinar un número entre 1 y 999, y el programa le informa en cada intento si el número introducido por el usuario es mayor o menor que el número que se busca. Adicionalmente, se cuenta el número total de intentos del usuario.

adivine_entero.py

```
# Juego en el que el usuario adivina un número

import numpy as np

# Genere un número aleatorio entre [1 y 1000)
```

```
number = np.random.randint(1,1000)

# Empiece el juego
guess   = int(input('Adivine número del 1 al 999: '))
guesses = 1

while guess != number:
    if (guess>number):
        print(guess," es muy alto")
    elif(guess<number):
        print(guess," es muy bajo")

    guess   = int(input("Adivine otra vez "))
    guesses = guesses + 1

print("\nExcelente, adivinó en ",guesses, ' intentos')
```

Un ejemplo al correr el programa es el siguiente:

```
Adivine número del 1 al 1000: 501
501  es muy alto
Adivine otra vez: 250
250  es muy alto
Adivine otra vez: 125
125  es muy alto
Adivine otra vez: 62
62  es muy bajo
Adivine otra vez: 90
90  es muy alto
Adivine otra vez: 75
75  es muy bajo
Adivine otra vez: 85
```

```
85  es muy bajo
Adivine otra vez: 88
88  es muy alto
Adivine otra vez: 87

Excelente, adivinó en  9 intentos
```

El programa inicia con la generación de un número aleatorio por medio de *np.random.randint(1,1000)*. Después, le solicita un número al usuario y, mediante un *while loop*, verifica si el número propuesto (*guess != number*) es igual al verdadero. Si no lo es, este informa si el número es mayor o menor, y vuelve a solicitarle al usuario otro número.

A continuación, se presenta una demostración de un programa que le pide un número real y positivo al usuario repetidamente. Si el número es negativo, se repite la solicitud (ya que solo queremos números positivos). Si el número es positivo, se calcula la raíz cuadrada con la función *sqrt* de NumPy. Si el usuario ingresa 0, el programa se detiene.

usuarioraiz.py

```
# Solicite al usuario un número real y
# tome la raíz cuadrada.
import numpy as np

x = None
while (x!=0.0):
    x = float(input("Digite un número real y positivo "))
    if (x<0.0):
        print("Número negativo")
        continue
    else:
        y = np.sqrt(x)
    print('sqrt(',x,') = ',y)
```

```
Digite un número real y positivo: 2.0
sqrt( 2.0 ) =  1.4142135623730951
Digite un número real y positivo: -1
Número es negativo
Digite un número real y positivo: 3.4
sqrt( 3.4 ) =  1.8439088914585775
Digite un número real y positivo: 0
sqrt( 0.0 ) =  0.0
```

El programa muestra el uso de un nuevo comando, *continue*, que le indica a Python que vaya directamente a la siguiente iteración; es decir, no va a imprimir el resultado *y*. En contraste, el comando *break* hace que el programa salga de un *for loop* por completo. ¿Puede escribir el mismo programa con un *for loop*?

4. Ejemplo: máximo común divisor

El máximo común divisor (o *greatest common factor* en inglés, GCF), de dos (o más) números enteros es el mayor número entero que los divide sin dejar residuo. Pero, ¿cómo se calcula?

- Se empieza con dos números, *a* y *b*.
- El GCF puede tomar valores entre 1 y el menor de los números *a* y *b*.
- La tarea es buscar entre todos los valores posibles, cuáles son un común divisor y cuál de estos es el mayor.
- Empiece con el *1*, que siempre será un común divisor.
- Siga con el *2*, *3*, etc. Si alguno de estos números no genera residuo, quiere decir que es un común divisor. Guárdelo como el GCF.
- Continue hasta *min(a,b)*.

Abajo, se expone un código para encontrar el GCF de dos números:

gcf.py

```
a = None
b = None
while (a!=0 or b!=0):
    txt = input('Digite dos enteros (0 0 termina)')
    a,b = txt.split()
    a   = int(a)
    b   = int(b)

    amin = min(a,b)
    if (amin<1):
        print('Sólo se aceptan números > 0')
        break

    mcd  = 1
    for j in range(2,amin+1):
        if (a%j==0 and b%j==0):
            mcd = j

    print('Máximo común divisor de',a,'y',b,'=',mcd)
```

A su vez, resultados para varios números:

```
Digite 2 enteros (0 0 termina) 3 2
Máximo común divisor =  1
Digite 2 enteros (0 0 termina) 7 21
Máximo común divisor =  7
Digite 2 enteros (0 0 termina) 9 24
Máximo común divisor =  3
Digite 2 enteros (0 0 termina) 923 1248
Máximo común divisor =  13
Digite 2 enteros (0 0 termina) 0 0
```

Vale la pena destacar que este programa no es muy eficiente si los números investigados son muy grandes, pero funciona muy bien para números pequeños. Algunas cosas nuevas en este programa son:

- *min(a,b)* - calcula el mínimo entre *a* y *b*, usando funciones internas de Python. También existe la función *max*. Esta puede tener más de dos argumentos de entrada.

- *a %i* - calcula el residuo de la división de los dos números (llamado el *módulo*). En este caso, *a %i=0* si *a* es divisible por *i*. Si el valor es diferente de cero, el valor obtenido es el residuo de dividir los dos números. Es importante recordar que el orden importa.

A continuación, se muestran otras funciones internas de Python disponibles (lista parcial):

```
abs(a)         Valor absoluto
float(a)       Conversión a número real
int(a)         Conversión a número entero
round(a)       Rodondear al entero más cercano
pow (x,y)      Calcular x a la y potencia
range(a,b,c) Secuencia de a hasta b, donde c es el salto
```

Problemas

1. Modifique el programa *gfc.py* para calcular el mínimo común múltiplo (*least common multiple* en inglés) de dos números enteros.

2. Escriba un programa que muestre una lista de números del 2 al 20. El programa debe calcular si el número es divisible por 2, por 3, por ambos, o por ninguno. La salida en pantalla puede tener el siguiente aspecto:

```
Num   Div por 2 y/o 3?
---   ----------------
 2         por 2
 3         por 3
 4         por 2
 5         ninguno
 6         ambos
 7         ninguno
```

3. Escriba un programa que repetidamente le solicite al usuario tres valores (reales) *a, b, c,* para la siguiente ecuación cuadrática:

$$a * x^2 + b * x + c = 0,$$

Usando la famosa fórmula para las raíces de este problema, el programa debe reconocer si hay raíces reales, y calcular estos valores. Como resultado, muestre el número de raíces reales y sus valores. El programa debe detenerse si todos los valores son 0.

4. Python incluye un módulo para la generación de números aleatorios (llamado *random*). Abajo se muestra un ejemplo que evidencia en pantalla 20 valores aleatorios entre 0 y 1.

rand_ej.py

```
# Genere una lista corta de números
# aleatorios entre 0 y 1
import numpy as np

for i in range(20):
   a = np.random.random()
   print (a)
```

Escriba un programa simple de Python para generar 10 000 números aleatorios entre 0 y 1. A continuación, realice un test de qué tan aleatorio es el generador de números, contando el número de veces que el

número cae dentro de 10 rangos distintos entre 0 y 1 (por ejemplo, 0 a 0.1, 0.1 a 0.2, etc.). Imprima el número total de veces para cada rango en la pantalla. Este es un ejemplo de qué aspecto puede tener la salida del programa:

```
0  1009
1  1048
2  1001
3  1038
4  1008
5   959
6   993
7   925
8  1017
9  1002
```

Ayuda: Cree un arreglo de números enteros (mire el módulo *numpy*) con 10 elementos, e inicie con ceros. Para cada número aleatorio, sume una unidad al rango apropiado, para así poder adicionar todos los números.

Nota: en C o Fortran, cada vez que corra el programa, se obtendrá la misma serie de números aleatorios. Esta no es una muy buena idea, pero el código para obtener series de números aleatorios distintas cada vez que se corre el programa es complicado. Python fija la semilla del número aleatorio (*seed*) con el reloj del computador, así que no debe ser un problema.

5. La Tierra, de manera general, se divide en varias *capas*, corteza, manto y núcleo (interno y externo). Si la Tierra tiene un radio de 6371 km, y lo comparamos con el radio de un balón de baloncesto (24.26 cm de diámetro), ponga en una tabla los datos, y compare:
   - Radio del núcleo interno
   - Radio hasta el tope del núcleo externo
   - Radio hasta el *Moho* (límite manto-corteza)

- Para todos los anteriores, determine su espesor.
- La atmósfera (la parte más densa) tiene 16 km de espesor. ¿Cuál es el espesor en nuestro ejemplo?

Espesores en km para cada capa:

$$T_{crust} = 25$$

$$T_{mantle} = 2900$$

$$T_{ocore} = 2250$$

$$T_{icore} = 1196$$

6. Calcule la presión P en el interior de la Tierra, a diferentes profundidades h. Asuma presión hidrostática para la corteza, una densidad del granito ρ, y que g (aceleración de la gravedad) es constante.

$$P = g * \rho * h$$

Capítulo 4

Funciones del usuario

def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min
gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gc
min = min amin = min amin = min amin = min amin = min ami
def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): def gcf(a,b): de
n amin = min amin = min amin = min amin = min amin = min

A medida que la longitud y complejidad de los programas de computador aumentan, se hace necesario dividir el problema en partes pequeñas. Esto constituye una buena estrategia, ya que lo hace más modular, y más fácil de leer y entender (y permite encontrar fácilmente problemas o *bugs*). Esto se puede hacer creando funciones (*def* o definiciones) dentro de Python, que hacen parte del trabajo. Algunas ventajas de programar así, son:

- Se pueden evaluar partes o pedazos del código de manera individual, confirmar que están funcionando de forma correcta antes de terminar el programa completo.
- El código es más modular y fácil de entender.
- Es más fácil usar partes del programa en otros programas (sin necesidad de tener múltiples copias del mismo código).

1. Funciones dentro del programa

Como ilustración de cómo introducir una función del usuario, el programa para adivinar un número de 1 al 1000 se repite, pero incluyendo una definición.

adivine_entero2.py

```
# adivina un número, usando funciones

def guess_number(number):
    """ Función para adivinar un número
    Función que hace el trabajo de interactuar
    con el usuario y evaluar si adivinó el número.
    Entrada: number - entero, que el usuario no conoce

    Salida: guesses - Número de intentos para adivinar
    """
    guesses  = 1
```

```
    guess    = int(input('Adivine un
                     número: 1 al 999: '))
    while guess!=number:
        guesses = guesses + 1
        if (guess>number):
            print(guess," es muy alto")
        elif (guess<number):
            print(guess," es muy bajo")
        guess = int(input("Adivine otra vez: "))

    return guesses

#-------------------------------------
# El programa principal
#-------------------------------------

import numpy as np

# Obtenga número aleatorio [1, 1000)
rnum = np.random.randint(1, 1000)

# Llame a la función que interactúa con usuario
nguess = guess_number(rnum)

print("\nExcelente, adivinaste en ",nguess, ' intentos')
```

```
Adivine un número: 1 al 999: 501
501  es muy bajo
Adivine otra vez: 750
750  es muy alto
...
```

```
Adivine otra vez: 583
583  es muy alto
Adivine otra vez: 581

Excelente, adivinaste en  10  intentos
```

Cabe anotar que el programa principal genera el número que queremos adivinar, pero la interacción con el usuario se realiza dentro de *guess_number*. La función o definición se escribe dentro del programa (en la misma celda), pero debe estar antes del programa principal.

```
def guess_number(number):
   ...
   return guesses
```

La variable *number* entra como argumento de la función *guess_number*, y el resultado de la función se da con la variable *guesses*, que representa el número de intentos del usuario. Note que debe usar *return guesses* para que el programa principal reciba de vuelta una variable al llamar la función. Tenga en cuenta asimismo que se requiere mantener la indentación dentro de la función.

El programa principal llama la función a través del siguiente comando:

```
nguess = guess_number(rnum)
```

De tal forma, la variable *number* dentro de la función tomará el valor que determina el programa al llamar a la función *rnum*. Tenga en cuenta, sin embargo, que, a diferencia de Fortran, si las variables son cambiadas dentro de la función, estas no cambian en el programa principal. Hay que tener cuidado con esto.

1.0.1. Máximo común divisor otra vez

El ejercicio del máximo común divisor (MCD) se puede separar en dos partes. La primera está encargada de la interacción con el usuario (pedirle los números, etc.). La segunda, de calcular el MCD. La última la podemos poner como una función.

gcf2.py

```
# MCD con una función

"""Cree la definición para calcular el MCD"""
def gcf(a,b):
    amin = min(a,b)
    mcd  = 1
    for j in range(2,amin+1):
        if (a%j==0 and b%j==0):
            mcd = j
    return mcd

# Ahora, el programa principal
for i in range(10):
    intxt = input('Digite dos enteros (0 0 termina) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
    if (a==0 and b==0):
        break
    mcd = gcf(a,b)  # debe crear la función

    print ("Máximo común divisor = ", mcd)
```

Observe que, en este caso, se define la función para calcular el MCD que recibe dos números de entrada (x,y), y devuelve al programa la variable *mcd*.

```
def gcf(x,y):
   ...
   return mcd
```

El número y la posición de los argumentos deben coincidir con los de la función; sin embargo, observe que los nombres de las variables no tienen que ser iguales ($x = a$).

2. Funciones con más salidas

Las funciones descritas en la sección anterior tienen una utilidad limitada, ya que están diseñadas para regresar al programa una sola variable como resultado. En algunos casos, se requiere de funciones que puedan regresar múltiples variables como resultado de una función. En Fortran, esto se lleva a cabo con subrutinas (*subroutine*). En Python, la misma función puede retornar distintas variables como resultado. Asimismo, en Python, la forma como se devuelven dichas variables puede ser diversa, incluyendo *listas*, *tuples*, etc., pero para facilitarla en este libro, se utilizará *tuples*.

A continuación, un ejemplo de programa en Python con utilidad en geociencias para calcular la distancia y azimuth de dos puntos sobre la superficie de la Tierra:

usuariodist.py

```
# Calcula la distancia y azimut entre dos
# puntos sobre una esfera.

def sph_azi(flat1,flon1,flat2,flon2):
    """
    SPH_AZI computes distance and azimuth between
    2 points on the sphere

    Inputs: flat1 = latitude of first point (degrees)
```

```
        flon2 = longitude of first point (degrees)
        flat2 = latitude of second point (degrees)
        flon2 = longitude of second point (degrees)

Returns: del = angular separation (degrees)
        azi = azimuth from 1 to 2, from N (deg.)

Notes:
(1) applies to geocentric not geographic lat,lon
on Earth
(2) This routine is accurate depending on the
precision of the real numbers used. Python
should be accurate to real(8) precision. For
greater accuracy, perform a separate calculation
for close ranges using Cartesian geometry.
"""

# importe módulos necesarios
import math  as mt
import numpy as np

if ( (flat1 == flat2 and flon1 == flon2)
or (flat1 == 90. and flat2 == 90.)
or (flat1 == -90. and flat2 == -90.) ):
    delta = 0.
    azi = 0.
    return [delta,azi]

# Perform calculation
delta = 0.
azi   = 0.
```

```
    raddeg=mt.pi/180.

    theta1=(90.-flat1)*raddeg
    theta2=(90.-flat2)*raddeg

    phi1=flon1*raddeg
    phi2=flon2*raddeg

    stheta1=mt.sin(theta1)
    stheta2=mt.sin(theta2)
    ctheta1=mt.cos(theta1)
    ctheta2=mt.cos(theta2)

    cang=stheta1*stheta2*mt.cos(phi2-phi1)+
        ctheta1*ctheta2
    ang=mt.acos(cang)
    delta=ang/raddeg

    sang = mt.sqrt(1.-cang*cang)
    caz  = (ctheta2-ctheta1*cang)/(sang*stheta1)
    saz  = -stheta2*mt.sin(phi1-phi2)/sang
    az   = mt.atan2(saz,caz)
    azi  = az/raddeg

    if (azi < 0.):
        azi=azi+360.

    return [delta, azi]

#-------------------------------------------------
# Programa principal
#-------------------------------------------------
```

```
lat1 = None
lon1 = None
lat2 = None
lon2 = None
while (lat1!=0. or lon1!=0. or lat2!=0. or lon2!=0.):
    intxt = input('Digite lat/lon del primer punto  ')
    lat1,lon1 = intxt.split()
    lat1      = float(lat1)
    lon1      = float(lon1)

    intxt = input('Digite lat/lon del segundo punto ')
    lat2,lon2 = intxt.split()
    lat2      = float(lat2)
    lon2      = float(lon2)

    delta,azi = sph_azi(lat1,lon1,lat2,lon2)
    print('Distancia y acimut:  %6.2f  %7.2f '
        %(delta, azi))
```

```
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon 40.71 -74.00
Distancia y acimut:   36.11     0.10
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon 48.85 2.35
Distancia y acimut:   77.63    40.91
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon -34.60 -58.37
Distancia y acimut:   41.90   160.50
Enter 1st point lat/lon 0 0
Enter 2nd point lat/lon 0 0
Distancia y acimut:    0.00     0.00
```

En este caso, los valores de *lat/lon* son pasados a la función *sph_azi*, la cual devuelve las variables *delta* y *azi*. Es importante siempre poner nombres a las funciones que sean claros, y que no repitan nombres de rutinas o variables ya existentes en Python.

También es esencial siempre documentar de forma clara lo que hace cada función, explicando las variables de entrada y de salida, y si la rutina tiene problemas de precisión, o cualquier otro potencial problema o limitación. Asimismo, aunque lo que se muestra para la función como documentación parece ser demasiada información, tenga en cuenta que en el futuro cualquier usuario va a tener una gran cantidad de funciones que puede utilizar, y es importante entender claramente qué hace cada función, y cómo se llama desde un programa. Esto puede ahorrarle al usuario mucho tiempo en el futuro.

De igual modo, la documentación busca además que el usuario o cualquier otra persona pueda utilizar la función sin necesidad de mirar el código. Por ejemplo, es claro que el azimuth es calculado desde el punto 1 al punto 2, y no al contrario.

Además, explicar las limitaciones de la función puede evitar que se use de manera errónea. En distancias muy cortas, el programa puede tener limitaciones, y devuelve un resultado que no es correcto, y el usuario puede no entender esto. La documentación advierte que otra función deberá ser usada.

Por último, note que la función busca ser robusta en casos particulares, como en cuanto a que los dos puntos tengan las mismas coordenadas (distancia y azimuth es cero), o estén en los polos.

3. *Modules* & *packages* propios

Para evitar reescribir mil veces la misma función, es posible escribir un *script* en el que se puede poner la función deseada, de tal forma que el programa principal pueda *importar* la función deseada, muy similar a cuando usamos:

```
import numpy as np
```

El siguiente ejemplo muestra el uso de la tercera ley de Kepler (que gobierna el movimiento de un satélite alrededor de un cuerpo primario - el Sol o un planeta). El archivo *kepler.py* es un archivo de texto plano, que contiene una sola función (definición), y que calcula el periodo de la órbita (duración de un giro alrededor del cuerpo primario) de un satélite, de acuerdo con la distancia con respecto al cuerpo (semieje mayor, para ser precisos) según la siguiente ecuación:

$$GM_{prim} = \frac{4\pi}{T^2}a^3,$$

donde G es la constante gravitacional, T es el periodo, a es el semieje mayor de su órbita, y M_{prim} es la masa del cuerpo primario.

kepler.py

```
def kepler_3ra(a,M=1.98847e30):
    """
    Función que usa la expresión de la 3ra ley de
    Kepler, para calcular el periodo para completar
    una órbita.
    Entradas:
       a - distancia en km del satélite con respecto
           al centro del cuerpo (planeta, Sol, etc)
       M - Masa del cuerpo (default - Masa del Sol)
    Salidas:
       T - Periodo para completar una órbita, en
           segundos
    """

    import numpy as np

    G = 6.67430e-11 # m^3kg^-1s^-2
```

```
    a_m = a * 1000.0   # in m

    T = np.sqrt(4.0 * np.pi**2 * a_m**3/(G*M))

    return T
```

La función (definición) en *kepler.py* muestra además cómo se asignan valores a variables opcionales. En este caso, la variable *M* tiene un valor predeterminado (*default*, que corresponde a la masa del Sol):

```
def kepler_3ra(a,M=1.98847e30):
```

El usuario, al llamar la función, puede usar:

```
kepler_3ra(a,M):
o
kepler_3ra(a):
```

En el segundo caso, la función automáticamente define *M*, asignándole el valor *1.98847e30*.

En tanto, para poder usar la función *kepler_3ra*, que se encuentra en el archivo *kepler.py*, se requiere *importar* el módulo, y, después, llamar la función, como en el ejemplo:

kepler_periodo.py

```
#    Programa que llama función propia de Kepler
#    para calcular el periodo de órbita de un satélite.
#

import kepler

d_sun   = 149.59e6 # distancia Tierra al Sol, en km
T_earth = kepler.kepler_3ra(d_sun)
```

```
T_days  = T_earth/86400

print('Periodo de órbita %6.2f días ' %(T_days))
```

En tal caso, el resultado es:

```
Periodo de órbita 365.22 días
```

El archivo que contiene la definición de la función es *kepler.py*, y puede tener otras definiciones de funciones. **Los módulos son entonces archivos de Python con una o muchas definiciones de funciones, clases, etc.**

3.1. Los *module*

Los *module* de Python son una forma natural para guardar definiciones de funciones que se usan de manera continua en este lenguaje de programación. De manera simple, los *module* son archivos de texto con extensión *.py* que contienen código de Python que puede ser importado y usado por otro programa de Python. En ese sentido, se pueden considerar como una librería, y son archivos que pueden tener funciones, variables, y mucho más.

Por ejemplo, un módulo de un programa puede enfocarse en la interacción con el usuario, mientras otro ejecuta la manipulación de datos (cargar datos), y otro hace los cálculos matemáticos requeridos. Entonces, todas las funciones que requieren interacción con el usuario se agrupan en un solo archivo *.py*, las funciones de carga de datos en otro *.py*, y al final las funciones que hacen cálculos, en un tercer *.py*. En tanto, para poder usar cada uno de los módulos, se deben importar en el programa principal con el comando *import*.

Al importar un módulo, se pueden usar las funciones adentro de este. Se pueden importar módulos internos de Python (como *numpy* o *math*), módulos de terceros, o módulos propios del usuario. Para nombrar los módulos, se recomienda usar nombres cortos, en minúscula, evitando usar símbolos especiales como punto (.) o interrogación (?). Evite usar por ejemplo *mi.modulo.py*, ya que esto puede interferir con la manera como Python importa los módulos.

Fuera de estas consideraciones, no se requiere nada especial para que un archivo sea considerado un módulo de Python; sin embargo, sí es importante recordar el mecanismo de importación de Python, de tal forma que su uso sea el adecuado.

Al usar el comando *import modu*, Python buscará el archivo *modu.py* en el directorio donde se está trabajando. Si no lo encuentra, Python buscará el archivo *modu.py* en el *path* definido por Python (ver más adelante cómo cambiar o adicionar directorios al *path*). Si no lo encuentra, se despliega un *ImportError*.

Si la importación es efectiva, Python ejecutará el módulo línea a línea. Es decir, si el módulo tiene comandos particulares, estos serán ejecutados, incluyendo comandos *import*. A su vez, las definiciones y clases serán guardadas en el diccionario del módulo.

Como resultado, todas las variables, funciones y clases que estén dentro del módulo estarán disponibles para el programa principal, utilizando el nombre del módulo (por ejemplo, *math.pi* es una variable del módulo *math*, y equivale a *3.1415...*). Este es un concepto fundamental de cómo trabaja Python.

En otros lenguajes (como Fortran o C), un comando *include file* se puede utilizar para que el programa lea el código dentro del archivo *file* y lo incluya (o copie) en el encabezado del programa principal. La ventaja de usar los módulos en Python, es que no existe el riesgo de que dentro del módulo haya una función que sobrescriba otra función con el mismo nombre de una función propia de Python.

Sin embargo, es posible cargar los nombres de funciones de manera directa, aunque eso puede ser **riesgoso**, así:

```
from math import *
```

Con este, se importan de manera directa las funciones del módulo *math*, como el coseno y el seno (ver ejemplos en el capítulo anterior). **Esto es considerado una** *mala práctica* **en Python**. Usar el comando *import* *, hace que entender el código y saber a qué módulo pertenece una función sea más difícil. Lo descrito no sucede con un comando como el siguiente:

```
from math import sin
```

Esta es una forma de indicar la función específica que se quiere importar, y asignarle un nombre global. Además, puede ser mejor que usar *import **, ya que muestra explícitamente qué es importado, de modo que la única ventaja es que al llamar la función se ahorra en digitación. Por supuesto, si se requiere cargar miles de funciones en un programa, el programa será muy difícil de leer.

Algunos ejemplos de importación de módulos como los siguientes no se recomiendan:

```
from modu import *
[...]
x = sqrt(4)  # de donde es sqrt
```

En este caso, ¿pertenece la función *sqrt* a *modu*? ¿O hace parte de las funciones propias de Python? ¿O tal vez la definió antes? El siguiente ejemplo es mejor, aunque no ideal para la importación:

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt ¿de modu?
```

En este segundo ejemplo, la función *sqrt* pertenece a *modu*, a menos que antes se haya redefinido. La opción recomendada es:

```
import modu
x = modu.sqrt(4)  # sqrt es de modu
```

En el último caso, es claro que *sqrt* pertenece a *modu*.

El objetivo de un buen estilo de programación no es escribir códigos cortos, sino códigos que sean legibles, claros y fáciles de utilizar por terceros (o por el mismo autor original, un año después). En tal marco, una mejor legibilidad

equivale a evitar exceso de texto o llenar espacio con comandos sin separación, pero eso tampoco significa que se deba llegar al extremo de oscurecer el código.

3.2. Los *packages*

En proyectos grandes, o para tener una *librería* de funciones, un programador busca agrupar funciones con diferentes objetivos. Python usa un sistema de **paquetes**, que es la extensión de módulos a un directorio. En resumen, un paquete es un directorio con uno o más módulos adentro.

Cualquier directorio con un archivo *__init__.py* es considerado por Python un paquete. A su vez, los módulos dentro del paquete pueden ser importados por un programa, de manera similar a los módulos individuales. El archivo *__init__.py* en principio tiene información y definición del contenido del paquete. Sin embargo, el archivo *__init__.py* puede estar vacío.

En tanto, un archivo *modu.py* en un directorio *pack/* puede ser importado con el siguiente comando:

```
import pack.modu
```

Este comando buscará un archivo *__init__.py* en el folder *pack/*, y ejecutará todos los comandos en el archivo. Después, buscará el archivo *modu.py*, y ejecutará sus comandos. Luego de esto, todas las variables, funciones y clases definidas en *modu.py* estarán disponibles con el nombre *pack.modu*.

Es también común ver que el archivo *__init__.py* tenga muchos comandos. Cuando un proyecto es complejo y grande, puede tener varios subpaquetes, y subsubpaquetes en una estructura de directorios larga y profunda. En ese caso, importar una función dentro de la subestructura, implicaría ejecutar muchos *__init__.py* durante la carga de directorio, subdirectorios y demás directorios dentro de la estructura.

Es usual dejar el archivo *__init__.py* vacío (sin nada escrito), e incluso esta es considerada una buena práctica. Sobre todo, es considerada una buena práctica si los módulos dentro de los paquetes y subpaquetes no requieren compartir

código (recuerde que Python ejecuta código línea por linea; por consiguiente, el orden en el que se importan los módulos importa).

Finalmente, es relevante el procedimiento adecuado para evitar tener códigos muy cargados de texto. Por ejemplo, si se quiere importar un módulo que se encuentra en un árbol de directorios complejo:

```
import pack1.subpack2.subsubpack3.modu
[...]
x = pack1.subpack2.subsubpack3.modu.sin(x)
```

En este caso, para usar la función *sin*, se requiere mucho texto, lo cual hace el código difícil de leer. Una mejor opción es nombrar las funciones dentro de un módulo con un emcabezado más corto o apodo (*mod*), como se muestra en el siguiente caso:

```
import pack1.subpack2.subsubpack3.modu as mod
[...]
x = mod.sin(x)
```

4. Ajustando el *path* para módulos propios

Cuando Python busca módulos, lo hace siguiendo el *path* predefinido en el sistema, incluyendo el directorio en el que está corriendo Python en ese momento. ¿Qué directorios están incluidos en el *path*? Esto se puede encontrar usando los comandos expuestos a continuación:

```
import sys
[...]
print(sys.path)
```

En esta situación, un ejemplo de respuesta es:

```
>>> ['',
'/opt/local/.../lib/python35.zip',
'/opt/local/.../lib/python3.5',
'/opt/local/.../lib/python3.5/plat-darwin',
'/opt/local/.../lib/python3.5/lib-dynload',
'/opt/local/.../lib/python3.5/site-packages']
```

Lo anterior significa que si el usuario quiere crear módulos propios o paquetes propios, estos deberían estar ubicados en alguno de los folders en el *path*. **En muchos casos, estos folders son del sistema, y no se recomienda cambiarlos o adicionarles archivos**.

En ese sentido, se recomienda crear un folder propio, en algún lugar en el cual el usuario pueda editar los archivos. Y, para que Python pueda encontrarlos, se debe adicionar el folder al *path*. Esto se puede hacer de dos maneras. La primera, es adicionar al *path*:

```
import sys
sys.path.append('/path/to/dir')
print(sys.path)
```

Así, el *path* quedará:

```
>>> ['',
'/opt/local/.../lib/python35.zip',
'/opt/local/.../lib/python3.5',
[...]
'/opt/local/.../lib/python3.5/site-packages'
'/path/to/dir']
```

Sin embargo, esta opción solo altera el *path* durante la ejecución del programa. La próxima vez que Python sea ejecutado, el *path* vuelve a su *default*.

Para adicionar el folder de manera permanente en el *path*, se debe adicionar la dirección del folder al *PYTHONPATH*. En sistemas operativos OS y Linux, esta adición se hace en el archivo *.bashrc*, así:

```
export PYTHONPATH="${PYTHONPATH}:/my/other/path"
```

En otros ambientes tipo Unix, esto se puede adicionar al archivo *bashrc*, *.profile*, *.cshrc*, o cualquiera que sea el archivo de inicio (*startup script*), dependiendo del *shell* que se use.

4.0.1. Windows

En ambiente Windows, para adicionar un folder al *path*, se requiere iniciar el *Anaconda Prompt*. Primero, es necesario saber si el entorno ya tiene definido el *PYTHONPATH*. Esto se puede ver digitando lo siguiente:

```
> set
```

A continuación, se busca en el resultado desplegado la variable de entorno *PYTHONPATH*. Cabe observar que el resultado, por lo general, está en orden alfabético. Asumiendo que el *PYTHONPATH* existe, y que no queremos borrar el contenido que tiene, para adicionar el folder debemos digitar lo siguiente:

```
> setx PYTHONPATH "%PYTHONPATH%;
    D:\usuario\folder1\folder2\"
```

La respuesta esperada de confirmación será:

```
CORRECTO: se guardó el valor especificado
```

En tanto, si el *PYTHONPATH* no existe, el comando es el siguiente:

```
> setx PYTHONPATH "D:\usuario\folder1\folder2\"
```

Problemas

1. Modifique el programa del problema 6 para calcular la presión litosférica, con una función propia dentro del programa.

2. Modifique el programa del problema 6 para calcular la presión litosférica, con una función propia en un módulo propio. Ponga el módulo en el directorio (paquete) que haya sido definido para módulos propios.

3. Cree una función propia (dentro de su *package* propio o dentro del programa) en Python, la cual calcule el volumen y la circunferencia de una esfera, si el usuario proporciona el radio. Haga un programa principal que le solicite el radio al usuario, y haga que este sea estable, que no acepte números negativos, y que imprima el resultado.

4. ¿Los astronautas experimentan *gravedad cero*? La siguiente ecuación (para un planeta esféricamente simétrico) muestra la aceleración de la gravedad $g = a_G$ de una masa *m*, debido a la masa de la Tierra, M_E, en función a la distancia r con respecto al centro del planeta. La aceleración está orientada hacia el centro de la Tierra.

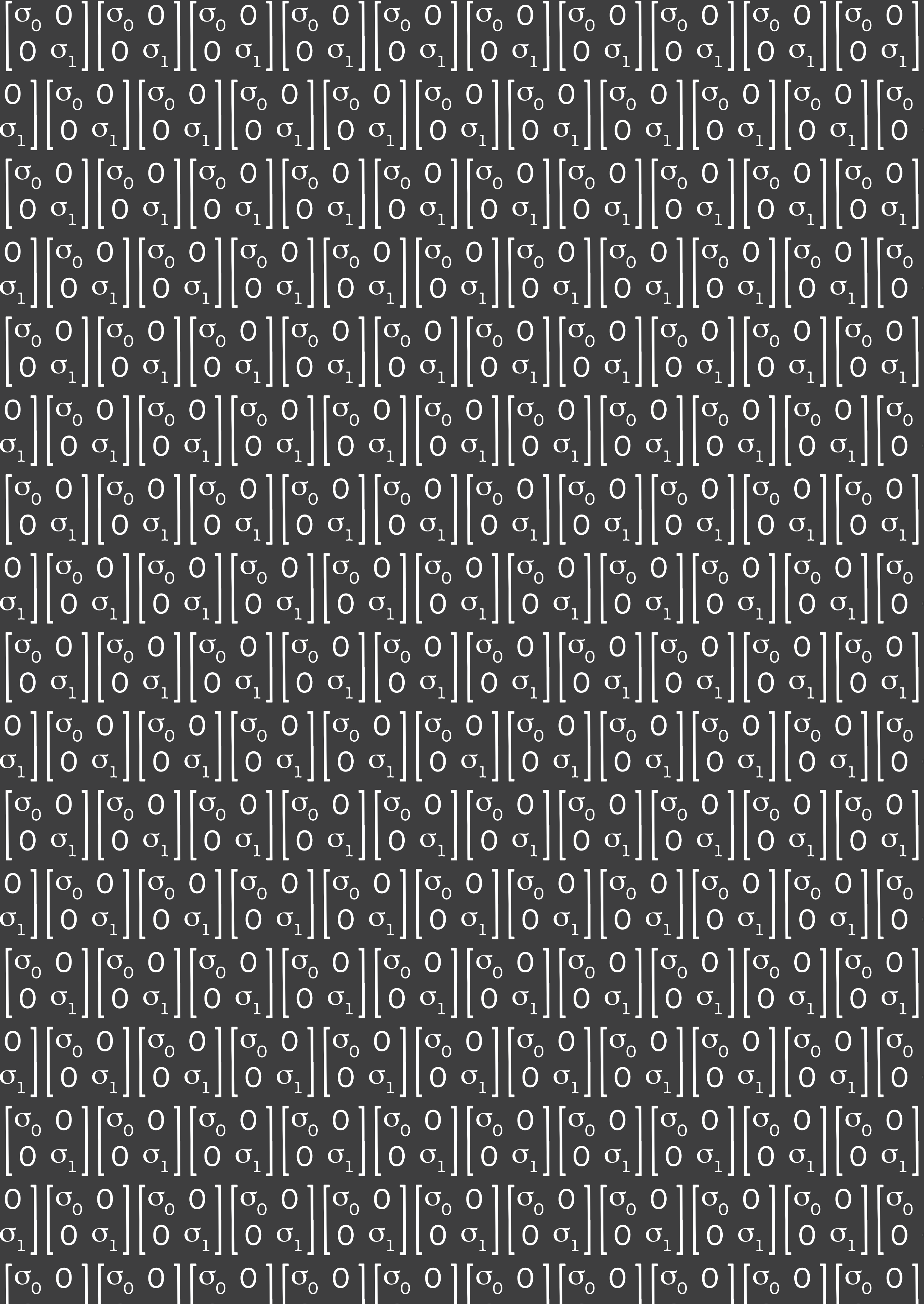
$$a_G = G\frac{M}{r^2}$$

Asumiendo la constante gravitacional $G = 6.6743 \times 10^{-11} m^3 kg^{-1} s^{-2}$, y que la masa de la Tierra $M_E = 5.972 \times 10^{24}$ kg, usando **funciones propias en lo posible**, calcule la aceleración en la superficie de la Tierra, y a *10, 100, 1000*, y *10 000* km de altura. También, defina la proporción de la aceleración para un astronauta a esas alturas, comparada con el caso de una persona que está en la superficie de la Tierra.



Capítulo 5

Arreglos: vectores y matrices



Hasta el momento, hemos trabajado con variables que representan un solo número (o conjunto de letras), pero en muchos casos se hace necesario trabajar con una lista de valores (un vector) o arreglos en 2D o 3D. En tanto, la multiplicación de vectores o matrices puede ser útil en métodos computacionales, y Python permite ejecutar tales operaciones.

En Python, hay varios tipos de estructuras de datos, llamadas *class*. Entre las más comunes están las *list*, *tuples* y *dictionaries*. De manera muy sencilla, estos se describen como:

list, como su nombre lo indica, es una lista de valores. Cada valor está numerado, empezando en cero; el primer valor está en la posición 0, el segundo en la posición 1, etc. Se pueden remover valores de una lista, adicionar valores a la lista, etc. Además, los valores dentro de una lista pueden ser de diferente tipo; por ejemplo, números y palabras.

tuples, que son similares a las listas, pero no se puede cambiar su valor. Los valores que se les ponen a los *tuples* no pueden ser cambiados dentro del programa. En tanto, la numeración es igual a la de las listas, empezando en cero. Un posible uso corresponde a los nombres de los meses del año, que no cambiarán.

dictionaries, como su nombre lo indica, es un diccionario. En un diccionario se tiene un índice de palabras, y, para cada nombre, una definición. En Python, la palabra se conoce como *key*, y la definición, como el *value*. Los valores del diccionario no están numerados, y no están ordenados de ningún modo específico. Además, se pueden adicionar, quitar o modificar los valores del diccionario. Un ejemplo es un directorio telefónico.

En muchas aplicaciones en geociencias, el objetivo es llevar a cabo un análisis numérico sobre valores de algún tipo, y, por esa razón, el uso de estas *classes* no es la más adecuada (esto incluye sumar valores a un vector, realizar multiplicación de matrices, rotación, etc.). Por esa razón, es necesario utilizar arreglos (numéricos o de otro tipo). En tal marco, los arreglos son como

listas, pero solo aceptan un tipo de entrada (números enteros, reales, o complejos, por ejemplo). Así, para la creación y el manejo de estos arreglos, vamos a utilizar los módulos de *NumPy*, que se importan a través de:

```
import numpy as np
```

Con esto, podemos crear arreglos numéricos de manera sencilla.

1. Arreglos numéricos

Vamos a crear los arreglos numéricos en Python con los módulos de *NumPy*, un paquete para análisis numérico.

```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(a)
```

```
[1 2 3 4 5]
```

De este modo, se define un arreglo a con 5 números del 1 al 5. Recuerde que en Python (como en C), el contador empieza en 0. Es decir:

```
a[0]
a[4]
```

```
1
5
```

1.1. Los números primos

Para empezar a dar uso a los arreglos numéricos, vamos a crear un programa que implementa arreglos para seleccionar números primos entre el 1 y el 100. Este es un ejercicio que nos permite pensar cómo se puede programar un

problema numérico. El objetivo es determinar si un número es primo o no, e imprimirlos. Sencillo, ¿no es verdad?

¿Qué es un número primo? Es un número natural mayor a 1 que tiene únicamente 2 divisores positivos distintos: él mismo y el 1. En palabras sencillas, son números enteros que no se puedan dividir por otro entero diferente de 1 y de sí mismos. Es decir:

```
2, 3, 5, 7, 11, 13, ...
```

Fíjémonos, por ejemplo, en que el número *4* no es primo, pues es divisible por *2*, o el *9* es divisible por *3*. En ese sentido, un método muy sencillo para definir los números primos es conocido como la *Criba de Eratóstenes*, en honor al matemático griego. En este método, se empieza con una lista de números del 1 al 100, y se considera que todos son primos, marcándolos con 0 (excepto el 1, por definición). Así, la estrategia es marcar todos los números que no son primos, poniendo el valor de 1 (lo que indica que no es un primo). ¿Cómo?

A continuación, se muestran tres columnas o arreglos, la primer columna representa el número (1, 2, ...), la segunda define si el número puede ser primo (0 para primo, *1* para no primo), y la tercera representa la posición de dicho número en e l arreglo (recordemos que Python empieza a contar de 0):

```
num  Primo? Posición
 1     1       0
 2     0       1
 3     0       2
 4     0       3
 5     0       4
 6     0       5
 7     0       6
 8     0       7
 9     0       8
 ...
```

```
 99     0       98
100     0       99
```

Por ahora, el único número que no es primo es el *1*. Se empieza con el número 2 (*num*=2), y se eliminan todos los múltiplos de 2, hasta el máximo que en este caso es 100 (*4, 6, 8, ...,98, 100*). El 2 es primo, ya que el valor en la segunda columna es 0.

```
num  Primo? Posición
 1      1        0
 2      0        1
 3      0        2
 4      1        3
 5      0        4
 6      1        5
 7      0        6
 8      1        7
 9      0        8
 ...
 99     0       98
100     1       99
```

Seguimos con el 3, un primo, ya que en la segunda columna sigue 0 (*num=3*), y se eliminan sus múltiplos (*6, 9, 12, ..., 96, 99*).

```
num  Primo? Posición
 1      1        0
 2      0        1
 3      0        2
 4      1        3
 5      0        4
 6      1        5
 7      0        6
```

```
 ccccv
 8      1       7
 9      1       8
 ...
 99     1      98
100     1      99
```

El 4 lo saltamos, ya que el 4 y todos sus múltiplos son múltiplos del 2, y ya han sido eliminados. Continuamos con el 5 (es un primo, el valor de la segunda columna es 0), y así sucesivamente.

El código para realizar este proceso es el siguiente:

primos.py

```
import numpy as np

maxnum   = 100
prime    = np.zeros(maxnum)
prime[0] = 1  # el 1 no es primo

maxi = int(np.floor(np.sqrt(maxnum)))
for ipos in range(1,maxi):
    if (prime[ipos]==0):
        inum   = ipos+1.      # 2, 3, 4, ...
        maxj = int(np.floor(maxnum/inum))

        for j in range(2,maxj+1):
            imult = inum*j      #  x2, x3, x4
            prime[imult-1] = 1  #  posición, i-1

nprime = 0
for ipos in range(maxnum):
    if (prime[ipos]==0):
```

```
        nprime += 1
        print("%4i" %(ipos+1))

print ("# primos encontrados ", nprime)
```

El resultado del código anterior es:

```
   2
   3
   5
...
  89
  97
# primos encontrados  25
```

Explicación

El programa comienza definiendo un arreglo *prime* lleno de ceros:

```
maxnum = 100
prime  = np.zeros(maxnum)
```

Luego, el primer número (*1*), de una vez, se define como *no primo*:

```
prime[0] = 1
```

El siguiente paso busca eliminar los números que no sean primos, usando dos *loops*. El primer *loop* evalúa la posición en el arreglo *prime*; si es 0, entonces dicha posición representa un número primo, así:

```
for ipos in range(1,maxi):
    if (prime[ipos]==0):
```

Si la posición representa un primo, se hace el segundo *loop* para eliminar todos sus múltiplos. Para esto, se define el número con $inum = ipos + 1$, y el *loop*

elimina todos sus múltiplos (*inum x 2, inum x 3, …*), asignando a la posición del arreglo *prime* el valor de *1*, así:

```
for j in range(2,maxj+1):
   imult = inum*j
   prime[imult-1] = 1
```

Cabe mencionar que los múltiplos se calculan con *inum*j*, pero su posición es *imult-1*, porque Python comienza a contar desde *j=0*. Es importante anotar además que solo se requiere evaluar todos los números hasta 10 (la raíz cuadrada de 100), porque los factores más grandes ya han sido eliminados. Las variables *maxi* y *maxj* se definen para no tener posiciones mayores a *maxnum=100* en este caso.

Cuando el programa termina de revisar los números hasta el 100, imprime el número de la posición en el arreglo *prime* que continúe siendo 0:

```
for ipos in range(maxnum):
   if (prime[ipos]==0):
        print("%4i" %(ipos+1))
```

Podemos contar los espacios que tienen 0 (total de primos entre 2 y 100), y además podemos saber qué números son primos (por su posición).

1.1.1. Los primos con funciones propias

El programa anterior imprime cada número primo en una línea, lo cual hace que el resultado sea muy largo si queremos más números primos (podemos imaginar lo que sucedería si resulta haber 1000 primos).

En ese sentido, la siguiente función (definición) retoma el código *primos.py*, pero busca los números primos, y devuelve únicamente los números primos encontrados en un arreglo. De esa manera, no imprime los resultados, solo retorna un arreglo con los números primos y el número total de números primos. El despliegue de los resultados debe ejecutarse en otro paso, con la ventaja de saber de antemano el número total de primos encontrados.

primos_subs.py

```
def primos_vector(maxnum):
    """
    prime_vector(maxnum)
    Función que busca números primos entre 2 y maxnum,
    y los ubica en un arreglo. El programa es muy lento
    si se buscan primos muy grandes.
    Entradas:
       maxnum - entero, buscar hasta maxnum

    Salidas:
       pvec   - vector con números primos, en orden
       nprime - primos encontrados
    """
    import numpy as np

    prime    = np.zeros(maxnum,dtype=int)
    prime[0] = 1

    max1 = int(np.floor(np.sqrt(maxnum)))
    for ipos in range(1,max1):
        if (prime[ipos]==0):
            inum   = ipos+1
            max2 = int(np.floor(maxnum/inum))
            for j in range(2,max2+1):
                imult = inum*j
                prime[imult-1] = 1

    # número de primos, y crear vector
    nprime = np.count_nonzero(prime==0)
    pvec   = np.zeros(nprime,dtype=int)
```

```
    pcnt = 0
    for ipos in range(maxnum):
        if (prime[ipos]==0):
            pcnt = pcnt + 1
            inum = ipos + 1
            pvec[pcnt-1] = inum

    return pvec,nprime
```

Así, esta nueva definición le devuelve al programa principal dos resultados, el vector con los números primos (*pvec*), y el número total de primos encontrados *nprime*. Esto quiere decir que el vector *pvec* tiene *nprime* números.

El programa *primos2.py* es modificado para que use la función *primos_vector*, y luego despliegue los resultados, imprimiendo el arreglo *primes* directamente:

primos2.py

```
# Programa para encontrar números primos
# llamando función propia, resultado en vector

primes,nprime = primos_vector(1000)
print ("# primos encontrados ", nprime)
print (primes)
```

El resultado del código anterior es:

```
# primos encontrados  168

[  2   3   ...  59  61
  67  71   ... 149 151
 ...
 829 839   ... 947 953
 967 971   ... 991 997]
```

O, si se prefiere, podemos modificar el programa para que se impriman solo 10 números por línea. El programa es el siguiente:

primos3.py

```
primes,nprime = primos_vector(1000)
print ("# primos encontrados ", nprime)

nprint = 0
for i in range(1,nprime):
      nprint = nprint + 1
      if (nprint%10==0 ):
         print ("%4i" % (primes[i]))
      else:
         print ("%4i" % (primes[i]),end="")
print('')
```

En este caso, el resultado del código es:

```
# primos encontrados  168
   3   5   7  11  13  17  19  23  29  31
  37  41  43  47  53  59  61  67  71  73
...
 881 883 887 907 911 919 929 937 941 947
 953 967 971 977 983 991 997
```

El código no cambia con respecto a *primos2.py*, excepto con lo que sucede al imprimir algunos números:

```
print ("%4i" % (primes[i]),end="")
```

En tal caso, el comando hace lo mismo que el comando *print*, pero no genera un salto de línea. El código revisa si ya se han desplegado 10 números primos (con el contador *nprint*), y permite que haya un salto de línea.

1.2. Arreglos 1D

Los valores de un arreglo se pueden asignar uno a la vez, o todos a la vez, con comandos sencillos, como en el siguiente programa:

testarreglos.py

```
import numpy as np

# Genere tres arreglos
x = np.array([1, 2, 3])
y = np.ones(3)
z = np.ones(3)*2

# Imprima los tres arreglos
print ('x = ',x)
print ('y = ',y)
print ('z = ',z)

# Asigne algunos valores
x[1] = 15
y[:] = 2
z = z-1

# Imprima otra vez arreglos
print('')
print('x = ',x)
print('y = ',y)
print('z = ',z)
```

Se da el siguiente resultado:

```
x =  [1 2 3]
y =  [1. 1. 1.]
```

```
z =  [2. 2. 2.]

x =  [ 1 15  3]
y =  [2. 2. 2.]
z =  [1. 1. 1.]
```

Debe tenerse en cuenta que cuando a un arreglo se le asigna un solo valor, cada elemento del arreglo toma ese valor (por ejemplo cuando hicimos *y[:] = 2*), pero es importante tener cuidado al ejecutar el comando *y=2*, pues la variable *y* queda definida como *int*; es decir, ya no es un arreglo o vector.

Para cambiar el valor de un elemento, se utiliza *x[1]=15* para modificar solo el valor del elemento en la posición 1 del arreglo. Debe considerarse que la posición dentro del vector o matriz, debe coincidir con el tamaño del arreglo. Si se usa una posición mayor a la disponible, Python arroja un error.

1.3. Arreglos 2D

Python puede generar arreglos en 2 dimensiones (o más) utilizando *NumPy*. A continuación, se presenta un programa que muestra algunas características de arreglos en 2D:

testmatriz.py

```
import numpy as np

y = np.empty([2, 3])
print('Empty Y, float ')
print(y)

x = np.zeros([2,3],dtype=int)
print('Zeros X enteros')
print(x)

x[0,:] = [1, 2, 3]
```

```
x[1,:] = [4, 5, 6]
print('X unidades')
print (x)

x = x + 1
print('X mas 1')
print(x)

c = np.array( [ [1,2], [3,4] ], dtype=complex )
print('Matriz compleja')
print(c)
```

Este tiene como resultado lo siguiente:

```
Empty Y
[[4.9e-324 9.9e-324 1.5e-323]
 [2.0e-323 2.5e-323 3.0e-323]]
Zeros X
[[0 0 0]
 [0 0 0]]
X unidades
[[1 2 3]
 [4 5 6]]
X mas 1
[[2 3 4]
 [5 6 7]]
Matriz compleja
[[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

Es importante tener en cuenta el orden como Python guarda la información de un arreglo, pues eso corresponde a lo que otras rutinas van a utilizar. Siempre se tiene *x(i,j)*, donde *i* son las filas y *j* son las columnas.

1.4. Aritmética en arreglos

La ventaja de utilizar los arreglos con *NumPy* es que además de guardar los datos en un formato de arreglos, se pueden realizar operaciones matriciales con ellos, algo que no es posible con *listas* o *tuples*. *NumPy* permite realizar operaciones con vectores y matrices, utilizando comandos propios, sin la necesidad de realizar la aritmética uno mismo. A continuación, vemos un ejemplo sencillo de algunas operaciones con vectores.

vectormath.py

```
import numpy as np

a = np.array( [1., 2., 3., 4., 5.] )
b = np.ones(5)*2
print("a     = ", a)

c = a + 1
print("a + 1 = ", c)

c = 2 * a
print("2 * a = ", c)

c = a * a
print("a * a = ", c)

print('')
c = np.sqrt(a)
print("sqrt(a) = ", c)

c = np.sin(a)
print("sin(a) = ", c)

c = np.exp(a)
```

```
print("exp(a) = ", c)

print('')
print("a = ", a)
print("b = ", b)

c = a + b
print("a + b = ", c)

c = a * b
print("a * b = ", c)

print('')
x = np.sum(a)
print("sum(a) = ", x)

c = a
c[3:5] = 0.0
print ("a con dos ceros al final", c )

x = np.dot(a,b)
print ("a dot b = ", x)

print('\nMedia Cuadrática')
rms = np.sqrt(np.sum((a-np.sum(a))**2)/len(a))
print ("rms = %6.2f " %(rms))
```

Observemos que es posible realizar operaciones complicadas con *NumPy*. En ese marco, es importante saber si el resultado es un escalar x o una matriz c. Y tengamos en cuenta que para operaciones como raíz cuadrada *sqrt()*, se debe utilizar la función perteneciente a *np*, y no la del módulo *math*, ya que la última solo trabaja con escalares. El resultado del programa anterior será:

```
a     =  [1. 2. 3. 4. 5.]
a + 1 =  [2. 3. 4. 5. 6.]
2 * a =  [ 2.  4.  6.  8. 10.]
a * a =  [ 1.  4.  9. 16. 25.]

sqrt(a)=  [1.         1.41 1.73 2.         2.23]
sin(a) =  [ 0.84  0.90  0.14 -0.75  -0.95]
exp(a) =  [  2.71  7.38   20.08  54.59 148.41 ]

a =  [1. 2. 3. 4. 5.]
b =  [2. 2. 2. 2. 2.]
a + b =  [3. 4. 5. 6. 7.]
a * b =  [ 2.  4.  6.  8. 10.]

sum(a) =  15.0
a con dos ceros al final [1. 2. 3. 0. 0.]
a dot b =  12.0

Media Cuadrática
RMS =   4.94
```

NumPy permite operaciones con matrices (transpuestas, producto punto, multiplicación matricial, etc.). A continuación, se muestran algunos ejemplos:

matrixmath.py

```
import numpy as np

a = np.array( [[ -5.1, 3.8, 4.2 ], \
              [ 9.7, 1.3, -1.3]] )

b = np.empty( [3,2])
b[:, 0] = [9.4, -6.2, 0.5 ]
```

```
b[:, 1] = [-5.1, 3.3, -2.2]

print("Matrix a")
print(a)
print("Matrix b")
print(b)

c = np.matmul(a, b)
print ("matmul(a,b)")
print (c)

c = np.matmul(b,a)
print ("matmul(b,a)")
print (c)

print("Valor max de a ", np.amax(a))
loc2 = np.argmax(a)
loc1 = np.unravel_index(loc2, np.shape(a))
print("Posición del max of a", loc1)
print("Max(a) con su posición", a[loc1])

c = a + np.transpose(b)
print("Matrix a + transpose(b)")
print(c)

print("a shape ", np.shape(a))
print("b shape ", np.shape(b))

print("a size  ", np.size(a))
print("b size  ", np.size(b))
```

```
Matrix a
[[-5.1  3.8  4.2]
 [ 9.7  1.3 -1.3]]
Matrix b
[[ 9.4 -5.1]
 [-6.2  3.3]
 [ 0.5 -2.2]]
matmul(a,b)
[[-69.4   29.31]
 [ 82.47 -42.32]]
matmul(b,a)
[[-97.41  29.09  46.11]
 [ 63.63 -19.27 -30.33]
 [-23.89  -0.96   4.96]]
Valor max de a  9.7
Posición del max of a (1, 0)
Max(a) con su posición 9.7
Matrix a + transpose(b)
[[ 4.3 -2.4  4.7]
 [ 4.6  4.6 -3.5]]
a shape  (2, 3)
b shape  (3, 2)
a size   6
b size   6
```

Arriba, se muestra el uso de algunas funciones de *NumPy* como *matmul*, *amax* y *argmax*, así como el cálculo de la transpuesta de una matriz y su tamaño. Observe que la multiplicación matricial *matmul(a,b)*, no equivale la multiplicación de los elementos de la matriz como resultado de *a*b*.

Es importante notar que multiplicar *matmul(a,b)* es correcto, dado que las dimensiones son *(2,3)* y *(3,2)*. El caso contrario también es posible. Si las dimensiones de las matrices no lo permiten, Python genera un error.

En tanto, la función *np.argmax* reporta el índice del valor máximo dentro de una matriz (reporta el primer valor máximo), pero no la posición del elemento en 2D, solo la posición dentro de la matriz, de acuerdo con el orden de lectura. Para obtener los dos parámetros (posición en fila y columna), se utiliza *loc1 = np.unravel_index(loc2, np.shape(a))*.

Como es de esperarse, hay funciones para *amin* y *argmin* para determinar el valor mínimo dentro la matriz y su posición. La página de referencia para todas las funciones en *NumPy* se encuentra en https://docs.scipy.org/doc/numpy/reference/.

2. Arreglos dentro de funciones

Suponga que se quieren analizar los datos dentro de un arreglo como el siguiente:

```
x = np.array( [1 , 2, ..., 100])
```

Este cuenta con 100 elementos. Para mejorar la legibilidad de su código, esto se hace dentro de una función en Python:

```
x2,xsum = analisis(x,n)
```

A la función *analisis* se le entrega el vector x, y su tamaño n, y se obtienen de vuelta otro vector y otras variables.

En el siguiente ejemplo, la función recibe un arreglo (una sola dimensión), y devuelve el vector, pero con el promedio removido (*demean*), el promedio y la varianza.

arr_trab.py

```
def arr_trab(x):
    """
    Función para análisis de unos datos en vector
    Input: x      = array of given size, with numbers
           n      = size of the array
```

```
    Output y     = demeaned array
           x_mu  = valor promedio de x
           y_var = variance of array
    """
    import numpy as np

    n     = np.size(x)
    x_mu  = np.mean(x)
    y     = x-x_mu
    y_var = np.var(y)

    return y,x_mu,y_var
```

array2fun.py

```
import numpy as np

mu    = 25
sigma = 3.0
a = np.random.normal(mu,sigma,5)
n = np.size(a)
print('Arreglo original ')
print (a)

[b,x_mean,x_var] = arr_trab(a)

print('Arreglo corregido (demeaned)')
print(b)
print("Prom(x) = ",x_mean)
print("Var(x)  = ",x_var)
```

Como resultado, obtenemos lo siguiente:

```
Arreglo original
[26.73  27.62 30.35 25.50  24.22]
Arreglo corregido (demeaned)
[-0.14  0.73  3.46385312 -1.38 -2.66]
Prom(x) =  26.88
Var(x)  =  4.31
```

No olvidemos que las entradas y salidas pueden incluir variables y arreglos.

3. Una aplicación a tsunamis

Los tsunamis pueden ser ocasionados por diferentes procesos geológicos como terremotos, erupciones volcánicas o movimientos en masa submarinos. El comportamiento de estas "grandes olas"[31] es diferente al de las olas generadas por los vientos, ya que su longitud de onda L es mayor que la profundidad del océano. Entonces, la velocidad de propagación de este tipo de olas es:

$$V = \sqrt{(gD)},$$

donde g es la aceleración de la gravedad, y D la profundidad del océano. Observemos que, en este caso, la velocidad no depende de la longitud de onda L.

A su vez, la longitud de onda y velocidad están relacionadas según:

$$L = VT,$$

donde T es el periodo de la onda.

Las grandes longitudes de onda de los tsunamis hace que la pérdida de energía sea muy baja. Cuando el tsunami se acerca a la costa, con profundidades menores, sufre cambios; la velocidad de propagación se hace menor, y el tsunami es más lento, pero el flujo de energía (F) permanece constante:

$$F = VE = \frac{1}{8}\rho g h^2 V,$$

donde E es la energía de la onda, h la altura de la ola, y ρ la densidad del agua ($\sim$1 kg/m^3). Esto implica que la altura de la ola debe aumentar.

El siguiente programa calcula la velocidad de propagación de un tsunami en mar abierto ($D = 5$ km). El periodo de la onda es de 30 minutos, y su altura inicial es de 1 m. Asumiendo que la disipación de energía es pequeña (flujo de energía constante), el programa también calcula la velocidad, longitud de onda y altura de la ola, a medida que se acerca a aguas más someras.

tsunami.py

```
import numpy as np

g     = 9.8  # m/s^2
depth = 5000 # m
T     = 1800 # s, 30 min
h     = 1    # m wave height
rho   = 1    # kg/m3

V = np.sqrt(g*depth)
L = V*T

F = 1/8 * g * V * rho * h**2

print('Para un tsunami')
print('   Prof. del océano  = %4.1f m' %depth)
print('   Periodo           = %4.1f min' %(T/60))
print('   Altura Ola (h)    = %4.1f m' %(h))
print('')

print('Velocidad (V)    = %4.1f km/h'%(V/1000*3600))
print('Long de Onda (L) = %4.1f km' %(L/1000))

# Estimar valores a varias prof.
```

```
depth = np.array((2000,1000,500,100,50,25,10))
V     = np.sqrt(g*depth)
L     = V*T
h     = np.sqrt(8*F/(V * rho * g))


ndep = len(depth)
print('')
print('Prof(m)  V(km/s)  L(km)   h(m)')
for i in range(ndep):
    print('   %4i    %5.1f  %5.1f  %5.1f'
          %(depth[i],V[i]/1000*3600,L[i]/1000,h[i]))
```

```
Para un tsunami
   Prof. del océano   = 5000.0 m
   Periodo            = 30.0 min
   Altura Ola (h)     =  1.0 m

Velocidad (V)     = 796.9 km/h
Long de Onda (L) = 398.4 km

Prof(m)  V(km/s)  L(km)   h(m)
   2000    504.0  252.0    1.3
   1000    356.4  178.2    1.5
    500    252.0  126.0    1.8
    100    112.7   56.3    2.7
     50     79.7   39.8    3.2
     25     56.3   28.2    3.8
     10     35.6   17.8    4.7
```

4. *Strings* y arreglos de caracteres

Aunque en geociencias no es fundamental el manejo de arreglos de caracteres, en algunos casos es importante saber manejar archivos que tienen números y caracteres. Las cadenas de caracteres (*strings*) corresponden a uno de los tipos (*classes*) más comunes en Python. Se pueden crear al encerrarlos en comillas (dobles o sencillas). Asimismo, como ya vimos al comienzo, una variable puede ser asignada a una clase *string*:

```
a = "¡Hola Mundo!"
b = "Python"
```

Cabe señalar que todo lo que esté entre comillas va a pertenecer al *string*, incluidos los espacios en blanco. Para determinar el número de caracteres en un *string*, utilizamos el siguiente procedimiento:

```
len(a)
len(b)
```

En este caso, se obtienen como resultados *12* y *6*, respectivamente. Cabe anotar que el número de caracteres incluye los espacios en blanco, sean estos ubicados en la mitad, al comienzo, o al final. Es decir, los espacios en blanco son contabilizados por Python como un caracter.

También es posible solicitar *substrings*, y crear nuevas variables por medio de concatenación:

```
a[0]      = '¡'
a[0:5]    = '¡Hola'
b[0:6]    = 'Python'
a[0:6]+b = '¡Hola Python'
```

Sin embargo, en Python, no es posible asignar caracteres a una subparte de una variable. Intentar eso, generaría el siguiente error:

```
a[0:5] = '¡Holo'
>>> TypeError: 'str' object does not support ...
```

En el próximo ejemplo, se muestran varias operaciones básicas con *strings* que pueden ser útiles, incluyendo concatenación, repetición, remoción de espacios en blanco, encontrar cadena de caracteres, etc.

char_oper.py

```
a = "¡Hola Mundo!"
b = "Python"

c = a[0:6] + b # concatenation
print(c)

c = b*2 # Repetition
print(c)

print(a[:5],a[6:]) # Range Slice

# Find characters
i = a.find("ello")
print (i)

# Remove blanks
c = "   "+a+b+"  "
print(c)
d = c.strip()
print(d)

# Split string, with whitespace delimiter
c = a.split()
print(c)
```

Un ejemplo de cómo esto puede ser útil en programación, se muestra en el caso del mejor jugador de fútbol:

futbol.py

```
# Programa para interactuar con strings

a = input("El mejor jugador de fútbol de la historia ")

if (a.find("onal")>-1 or a.find("ich")>-1
    or a.find("rist")>-1):
   print("El portugués me gusta pero, y ¿Pelé?")
elif (a.find("Leo")>-1 or a.find("essi")>-1):
   print("Lo único que le falta es el Mundial")
elif (a.find("arad")>-1 or a.find("iego")>-1 or
      a.find("rmand")>-1 or a.find("elus")>-1):
   print("De acuerdo, el de la Mano de Dios")
else:
   print("Claro, Pelé siempre será el mejor")
```

Observemos que acá se espera un conjunto de respuestas (Cristiano, Messi o Maradona), para lo cual se responde de acuerdo a los tres posibles jugadores. En cambio, si el usuario pone una respuesta distinta, solo se responde de una manera; no importa quién sea el jugador, se asume que el usuario digitó Pelé.

4.1. Arreglos de caracteres

Como se mostró arriba, un *string* es tan solo una cadena de caracteres. Por ello, un arreglo 1D de *strings* sería en realidad un arreglo 2D. A diferencia de lo que se puede hacer en Fortran, con *NumPy* no es fácil crear arreglos de caracteres, y en cambio se usan listas, *list*.

Un ejemplo del uso de listas en Python es el siguiente:

lista_caracter.py

```
# lista_char.py
# Uso de listas de strings de caracteres

lista = ['Mercurio', 'Venus', 'Tierra', 'Marte',
         'Júpiter', 'Saturno', 'Urano','Neptuno']
print ('Nombres en la lista: ', len(lista))

print("El 2do planeta del Sistema Solar es ", lista[1])
print("El primer planeta gaseoso es   ", lista[4])

lista.append("Plutón")

print(lista[-1], "era considerado un planeta")
print ('Nombres en la lista: ', len(lista))
```

Cabe mencionar, que la cantidad de caracteres en cada valor de una lista puede ser diferente. En Python, se pueden utilizar algunos métodos sobre las listas (sean de caracteres o números, o ambos).

```
list.append(elem) -- adiciona un elemento a la lista
list.insert(index, elem) -- inserta un elemento,
        corre los demás a la derecha

list.extend(list2) -- pega list2 a list,
        Se puede usar + o +=
list.index(elem) -- búsqueda y posición de un elemento,
        Error si no existe.
```

Otras funciones sobre listas incluyen *list.remove()* y *list.pop()* para retirar elementos, o *list.sort()*, *list.reverse()* para reordenarlos, entre muchos más.

Problemas

1. Basado en el programa *primos.py* y la función *enc_primos.py*, escriba un programa que genere una matriz en la cual se organicen los números primos. Pero, en este caso, organícelos en forma de hélice, es decir:

```
2x2 matrix
[[ 2.  3.]
 [ 7.  5.]]
3x3 matrix
[[ 17.  19.  23.]
 [ 13.   2.   3.]
 [ 11.   7.   5.]]
4x4 matrix
[[ 17.  19.  23.  29.]
 [ 13.   2.   3.  31.]
 [ 11.   7.   5.  37.]
 [ 53.  47.  43.  41.]]
```

   Permita que se pueda crear la matriz de diferentes tamaños (5x5, 6x6, etc.).

2. Un análisis simple de una serie de datos incluye el cálculo del promedio (*mean*) y la desviación estándar (*std*). Sin embargo, algunos datos tienen distribuciones estadísticas que no son normales (*gaussianas*), y una mejor descripción del valor medio de los datos se obtiene con la mediana (*median*). Por tanto, para calcular la mediana, se requiere organizar los datos de manera ascendente. Con tal objetivo, abajo mostramos una función que permite reorganizar un arreglo de números utilizando el método de *quicksort*:

qsort.py

```
def quicksort(arr_in):
    """
    Función que realiza un quicksort
    Entradas
        arr_in - arreglo con números, no organizado
    Salidas:
        arr     - arreglo, pero organizado
    """

    import numpy as np

    if (arr_in.ndim>1):
        print("Arreglo con dimensiones erradas")
        return None

    # Make a copy of the array
    arr = np.copy(arr_in)
    n   = arr.size

    for j in range(2,n+1):
        ibreak = 0
        A = arr[j-1]
        for i in range(j-1,0,-1):
            if (arr[i-1] <= A):
                arr[i] = A
                ibreak = 1
                break
            arr[i]=arr[i-1]
        if (ibreak==0):
            arr[0] = A
    return arr
```

Genere dos arreglos de 50 y 51 números aleatorios con *NumPy*, y con funciones propias determine el promedio, la desviación estándar y la mediana del arreglo. No utilice funciones Python o de *NumPy* para el cálculo. Compare sus resultados con las funciones *np.mean* y *np.median*.

Para generar su arreglo, puede usar el siguiente procedimiento:

```
C = np.random.normal(0,1,51)
```

3. En geología estructural, tectónica o sismología, muchas veces, se estudian los esfuerzos, y se describen mediante el tensor de esfuerzos. Supongamos que tenemos un tensor de esfuerzos horizontales (solo de dos dimensiones):

$$\tau = \begin{bmatrix} \tau_{xx} & \tau_{xy} \\ \tau_{yx} & \tau_{yy} \end{bmatrix}$$

Para un tensor de esfuerzos, siempre es posible encontrar un sistema de coordenadas, de tal forma que no haya esfuerzos de cizalla ($\tau_{xy} = \tau_{yx} = 0$), de modo que se identifican los ejes principales de esfuerzo:

$$\tau^R = \begin{bmatrix} \tau_1 & 0 \\ 0 & \tau_2 \end{bmatrix}$$

Por convención, los ejes principales están organizados de tal forma que

$$|\tau_1| > |\tau_2|$$

Escriba un programa para determinar el ángulo qué debe rotar el tensor de esfuerzos τ, a saber:

$$\tau = \begin{bmatrix} -27.0 & -7.1 \\ -7.1 & -13.0 \end{bmatrix}$$

Así, buscamos obtener (lo más cercano posible a) un tensor $\tau^{\mathbf{R}}$ con los esfuerzos de cizalla $\tau_{xy} = \tau_{yx} = 0$. Para rotar el tensor un ángulo α debe realizar la operación matricial:

$$\tau^R = R\tau R^T,$$

donde

$$R = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

Recordemos que el ángulo debe estar expresado en radianes. Permita que el programa haga la búsqueda del ángulo; por ejemplo, con un *for loop*.

Nota: el mismo (y mejor) resultado se puede obtener utilizando algoritmos de algebra lineal; por ejemplo, utilizando los valores y vectores propios (*eigenvalues* y *eigenvectors*) de la matriz. Sin embargo, esto está por fuera del enfoque de este texto.



Capítulo 6

Lectura y generación de archivos

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5
STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2
STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2
... ...

7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7.90 -76.5 STA1 7
9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9.78 -73.2 STA2 9
2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2.02 -78.2 STA3 2
... ...

Hasta ahora, todos los ejemplos en el libro usan entradas del usuario con el teclado, y, como salida, se despliega en pantalla el resultado. Esto es útil para trabajos pequeños, pero cuando se quiere analizar grandes cantidades de datos, es relevante poder importarlos a Python. Estos datos pueden estar en tablas de Excel, de Matlab, o en archivos de texto plano (ASCII), o pueden provenir de un equipo de campo o del laboratorio. En este capítulo, se busca entender cómo abrir, leer y escribir series de datos. Python tiene paquetes para leer datos de varias fuentes, incluyendo Excel, Matlab y otros, pero para el objetivo de este libro, solo nos concentraremos en leer archivos de texto plano. Tengamos en cuenta que tanto Matlab como excel pueden exportar los datos en dicho formato.

1. Cómo leer un archivo de texto

Supongamos que se tiene un archivo de texto plano llamado *test_file.dat* que tiene las siguientes líneas:

test_file.dat

```
La primera línea
La 2da línea
Es la tercera
Cuarta línea
```

Tengamos en cuenta que en realidad a Python no le interesa la extensión del archivo (puede ser *.dat*, *.txt*, *.doc*, o incluso no tener una extensión). La forma como es leído el archivo depende del código.

Hay varias formas de leer este archivo: *1)* línea por línea, *2)* como una lista, *3)* asignarle una variable.

read_test.py

```
fname = 'test_file.dat'

# línea por línea
```

```
print('Lectura de archivo: línea por línea')
f = open(fname, 'r')
for line in f:
   print(line, end='')
f.close()

# poner en una lista
print('')
print('Lectura de archivo: en una lista')
f = open(fname, 'r')
f_list = list(f)
print(f_list)
f.close()

# lectura completa
print('Lectura de archivo: en una variable')
f = open(fname, 'r')
text = f.read()
print(text)
```

Tengamos presente que *fname* debe ser un archivo que se encuentra en el directorio donde se está trabajando, o debe tener la dirección correcta con la ubicación del archivo.

El ejemplo anterior muestra varias de las funciones necesarias para leer archivos. Primero, se debe abrir, usando *open(fname)*, el archivo con nombre en la variable *fname*:

```
f = open(fname, 'r')
```

El resultado del comando anterior es una variable *f*, que para definirla debemos proporcionar el nombre del archivo (*fname*) que se quiere abrir, y la opción *'r'* indica que este archivo es solo de lectura (*read*). Se pueden usar otras opciones de lectura como *'w'* para escribir un archivo (si el archivo existe, será

sobrescrito), mientras *'a'* adiciona al archivo al final, y *'r+'* abre el archivo para leer y escribir. Para archivos binarios, se usa *'b'*. El modo de lectura es opcional; si no se utiliza, Python asume *'r'*.

En la primera forma de lectura, el archivo se lee de manera secuencial (línea por línea) con el siguiente procedimiento:

```
for line in f:
   print(line, end='')
```

Así, se imprime cada línea. Python lee el archivo línea por línea, hasta llegar a la última, y se detiene el *for loop*.

En el segundo tipo de lectura del archivo, se lee el archivo completo, y cada línea se pone en un elemento de una lista.

```
f_list = list(f)
```

Por último, está la forma de leer el archivo completo e imprimirlo con el siguiente comando:

```
text = f.read()
print(text)
```

El resultado final del código anterior es el siguiente:

```
Lectura de archivo: línea por línea
La primera línea
La 2da línea
Es la tercera
Cuarta línea

Lectura de archivo: en una lista
['La primera línea\n', 'La 2da línea\n',
   'Es la tercera\n', 'Cuarta línea\n']

Lectura de archivo: en una variable
```

```
La primera línea
La 2da línea
Es la tercera
Cuarta línea
```

2. I/O de datos en Python

Para el estudio de datos en geociencias, es necesario poder leer (y guardar) archivos en formato plano (*flat file*). En general, esto implica una tabla con datos en filas y columnas, con valores numéricos. En algunos casos, el archivo tiene un encabezado o *header* que no tiene valores numéricos, sino texto que explica el significado de cada columna.

Para archivos en otros formatos, por ejemplo datos binarios como *segy*, *mseed* o cualquier otro formato propio de cada subcampo de las geociencias, es necesario utilizar herramientas adicionales, o saber de forma precisa el formato, para poder leerlo. La lectura de dichos archivos no será estudiada en este libro.

2.1. Lectura de archivos

El archivo *some_data.dat* tiene una serie de datos organizados en tres columnas (con un número indeterminado de filas) separadas por espacios, así:

some_data.dat

```
  0.00  1.000  1.000
  0.20  0.990  0.977
  0.40  0.960  0.910
  0.61  0.910  0.804
...
 19.60  0.180 -0.139
 19.80  0.177 -0.118
 20.00  0.167 -0.086
```

El siguiente programa puede leer el archivo, y pone las tres columnas en una matriz *data* que luego puede ser analizada, o, como en este caso, desplegada en una figura.

read_data.py

```
import numpy as np
...

# nombre del archivo
fname = 'data/some_data.dat'

"""Cargue el archivo con Numpy"""
data = np.loadtxt(fname)

# Figura de columnas 1 vs 2, 1, vs 3
...
```

La figura 6.1 muestra el resultado. En el siguiente capítulo, veremos en detalle el uso de gráficas en Python; por ahora, se muestra el resultado de graficar la primera columna contra la segunda y la tercera. Los datos muestran la función Bessel de orden cero, que se usa para estimar la velocidad de propagación de ondas sísmicas en la corteza terrestre [27, 10].

Observe cómo *NumPy*, de forma automática, asigna el tamaño de la variable *data*, sin necesidad de que el usuario le indique cuántas filas tiene el archivo; sin embargo, es importante tener **precaución** pues el comando *np.loadtxt* asume que el archivo está organizado en columnas, separadas por espacios, y que todas las columnas están ocupadas con valores numéricos, tienen la misma longitud, y no tienen espacios vacíos.

El comando *np.loadtxt* tiene una gran variedad de opciones para leer los archivos (digite *help(np.loadtxt)* en Python para ver la documentación). El formato general de la función tiene muchas opciones (que se pueden consultar acá). En este caso, solo se muestran algunas opciones:

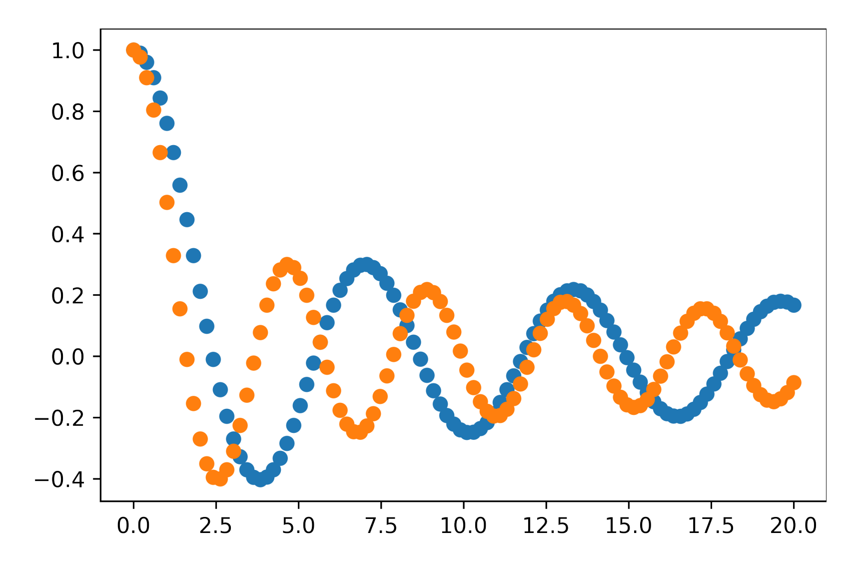


Figura 6.1. Resultado de lectura de archivo *some_data.dat*.

```
loadtxt(fname, dtype='float', comments='#',
        delimiter=',', skiprows=1, usecols=[0,2]
       )
```

Arriba, se le ordena a *loadtxt* leer los datos como *floats*, como números reales. En tanto, líneas con comentarios se marcan como #, y no son leídas, a la vez que las columnas están separadas por comas (típico de archivos *.csv*), se salta una línea que puede ser el encabezado, y solo se leen la primera y tercera columna. Cabe anotar que todos los comandos a excepción del nombre del archivo, son opcionales, y Python tiene valores predeterminados para las variables opcionales. Por ejemplo, las columnas por defecto están separadas por espacios, y los valores se asumen como *float*.

Veamos un segundo archivo, *some_data_header.dat*, que tiene en la primera línea un encabezado como el siguiente:

some_data_header.dat

```
#   theta       cos       sin
 0.00e+00  1.02e+00 -1.88e-02
 1.00e+00  9.95e-01  2.84e-03
 2.00e+00  9.88e-01  5.13e-02
...
 7.20e+02  9.87e-01 -2.48e-03
```

En este caso, la lectura del archivo se puede llevar a cabo *saltando* el encabezado, como se muestra a continuación:

read_data2.py

```python
import numpy as np

# nombre del archivo
fname = 'data/some_data_header.dat'

# Cargar datos
data = np.loadtxt(fname,skiprows=1)

y = data[:, 2]/data[:,1]

# Figura
...
```

2.1.1. Archivos más complejos

Hay archivos más complejos cuya lectura requiere mayor trabajo. Un ejemplo es un archivo que contiene los identificadores de las estaciones de la Red Sismológica Nacional de Colombia (RSNC), incluyendo latitudes, longitudes y elevaciones (se puede consultar en sgc.gov). Dicho archivo tiene el siguiente formato:

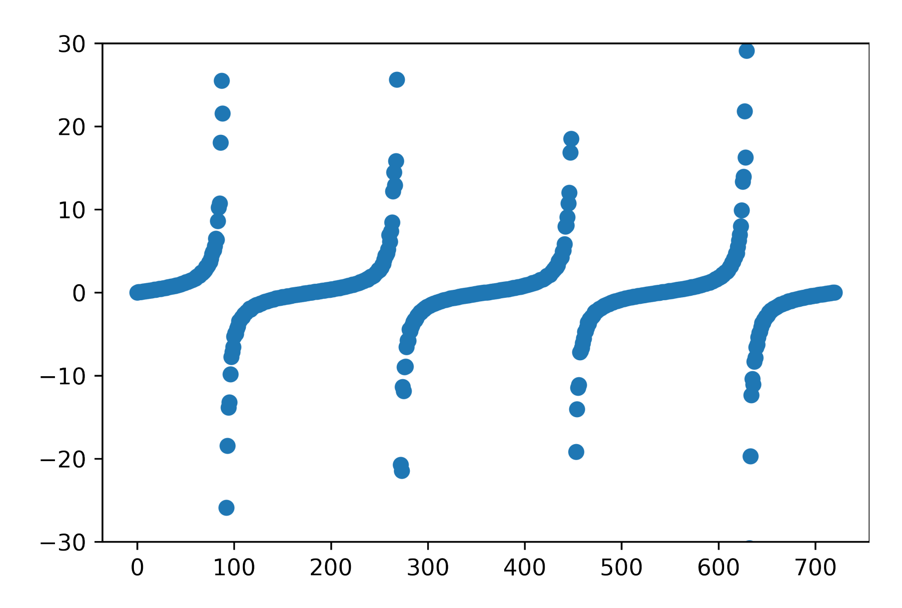


Figura 6.2. Resultado de lectura de archivo *some_data_header.dat*, dividiendo la tercera columna por la segunda (*función tangente*).

```
APAC            7.9     -76.58       195
ARGC            9.585   -74.246      117.9
BBAC            2.022   -77.247      1716
...
YPLC            5.397    -72.38       696
YOT             3.983   -76.345      1059
ZAR             7.492   -74.858       200
```

En este ejemplo, las diferentes columnas están separadas por espacios (el número de espacios puede variar).

NumPy tiene una función que en principio puede leer este tipo de archivos *np.genfromtxt()*. Sin embargo, aquí presentamos el paquete *Pandas*. Este paquete tiene la capacidad de leer archivos planos, mixtos (caracteres y números), e incluso archivos de Excel y Matlab. A continuación, un ejemplo de cómo leer el archivo anterior:

read_stations.py

```
import numpy as np
import pandas as pd

fname = 'data/rsnc.dat'
data = pd.read_csv(fname, delim_whitespace=True,
         header=None)
print('Estructura de Pandas')
print(data)

sta = data.iloc[:,0].to_numpy()
lat = data.iloc[:,1].to_numpy()
lon = np.array(data[2])

# Mapa de los resultados
...
```

donde se muestra la estructura de *Pandas*.

```
Estructura de Pandas
       0       1       2       3
0   APAC   7.900 -76.580   195.0
1   ARGC   9.585 -74.246   117.9
2   BBAC   2.022 -77.247  1716.0
..   ...     ...     ...     ...
66   YOT   3.983 -76.345  1059.0
67   ZAR   7.492 -74.858   200.0
[68 rows x 4 columns]
```

Fíjémonos en que para *convertir* los datos con la estructura de *Pandas* a la estructura de *NumPy*, hay varias opciones:

```
sta = data.iloc[:,0].to_numpy()
lat = data.iloc[:,1].to_numpy()
lon = np.array(data[2])
```

Lo anterior arroja tres arreglos para *sta*, *lat* y *lon*.

```
['APAC' 'ARGC' 'BBAC' ...
...  'YOT' 'ZAR']
[ 7.9    9.585  2.022  ...
 ...  3.983  7.492]
[-76.58  -74.246 -77.247 -...
...  -76.345 -74.858]
```

En ese contexto, el comando *data.iloc[:,0].to_numpy()* toma la primera columna de la variable *data*, incluyendo todas las filas. Algo similar se puede obtener con el comando *np.array(data[0])*; sin embargo, puede resultar más claro el primer comando, con el cual se define de manera clara cuál fila o columna se desea, y se puede lograr que lo convierta a un arreglo *NumPy*.

Aunque no se muestra el código, la figura 6.3 genera un mapa con la ubicación de las estaciones en Colombia.

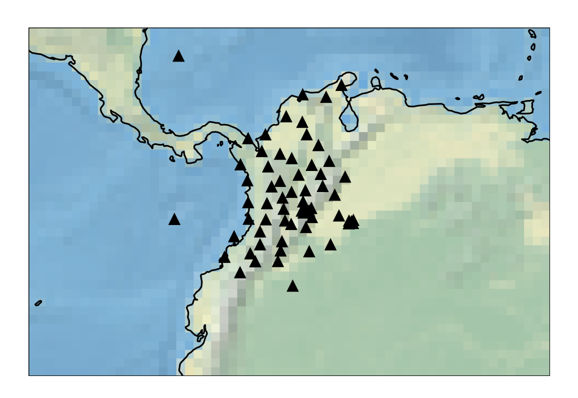


Figura 6.3. Resultado de lectura de archivo *rsnc.dat*. Despliegue de la ubicación de las estaciones de la Red Sismológica Nacional de Colombia (RSNC).

Es importante tener en cuenta que se debe importar el paquete *Pandas*, así:

```
import pandas as pd
```

Además, para cargar los datos, se usa el siguiente comando:

```
data = pd.read_csv(fname, delim_whitespace=True,
        header=None)
```

De este modo, se le indica al código que no hay encabezado, y que los separadores entre columnas están marcados por espacios libres (pueden ser tabs, comas, etc.). Lo curioso de *Pandas* es que tiene su propia estructura (llamada *DataFrame*), que es una especie de arreglo, pero que puede tener diferentes tipos de elementos (*str, int, float*, etc.).

2.2. Guardar archivos

En muchos casos, es importante poder leer archivos, pero también es fundamental poder guardar los resultados de un análisis, de procesamiento o de simulaciones y cálculos, sin la necesidad de repetir el procesamiento cada vez que se quiera revisar los resultados. Aunque las gráficas son una forma de presentar resultados, no permiten mirar los datos crudos, que siempre es bueno tener disponibles para poder replicar trabajos de investigación.

En el siguiente ejemplo, se lee un archivo *another_data.dat*, que tiene dos columnas (con pares de números):

another_data.dat

```
  3.6809  -1.6855
  3.4113  -2.5989
...
  6.6905  -3.0042
```

El objetivo es realizar un procesamiento de los datos, y guardar un nuevo archivo *test.dat* con el resultado. El código debe:

- Leer el archivo *another_data.dat*.
- Para cada fila, calcular el producto de las dos columnas.
- Guardar un nuevo archivo *test.dat* con las dos columnas originales, y una tercera columna con el producto.

Es importante tener cuidado, ya que esto puede sobreescribir un archivo ya existente.

fileinout.py

```
import numpy as np

fin  = 'data/another_data.dat'
fout = 'dout/test.dat'

# Cargar los datos, dos arreglos x y y
data = np.loadtxt(fin)
x = data[:,0][:,None]
y = data[:,1][:,None]

# producto de los dos arreglos
z = x*y

# pegue arreglos en una sola matriz
M = np.hstack((x,y,z))

# Guarde un nuevo archivo, con formato
np.savetxt(fout, M,fmt='%5.2f %5.2f %5.2f')
```

El archivo de entrada se abre con el siguiente comando:

```
data = np.loadtxt(fin)
```

En este caso, la variable *fin* es el nombre del archivo que se quiere abrir. Si el archivo no existe, Python generará un error. Posteriormente, los valores dentro del archivo son leídos con el siguiente comando:

```
data = np.loadtxt(fin)
x = data[:,0][:,None]
y = data[:,1][:,None]
```

Adicionalmente, las dos columnas son separadas en las variables *x* y *y*. En este ejemplo, se presenta una notación nueva, con la cual *x[:,None]* busca generar un arreglo vertical (un vector) con tamaño *(21, 1)*. Si esto no se hace, el arreglo tendría la forma *(21,)*, y no permitiría hacer operaciones matriciales. A su vez, la función *loadtxt*, como se vio en la sección anterior, identifica qué tan largo es el archivo. Después, se hace la multiplicación de los dos arreglos *z=x*y*. Para facilitar guardar el nuevo archivo, se genera un arreglo 2D:

```
M = np.hstack((x,y,z))
```

De esta forma, los tres arreglos 1D se juntan, pegándolos horizontalmente; es decir, cada arreglo representa una columna. La matriz resultante *M* tiene un tamaño *(21,3)*. Los tamaños se pueden confirmar del siguiente modo:

```
print('Tamaño arreglos X, Y, X*Y')
print(np.shape(x), np.shape(y), np.shape(z))
print('Tamaño arreglo M')
print(np.shape(M))
```

Y se obtiene:

```
Tamaño arreglos X, Y y X*Y
(21, 1) (21, 1) (21,1)
Tamaño arreglo M
(21, 3)
```

Para concluir, esa nueva matriz *M* se guarda en el archivo:

```
np.savetxt(fout, M,fmt='%5.2f %5.2f %5.2f')
```

El archivo resultante tendrá un formato específico. Cabe anotar que el comando de formato *fmt* es opcional y, si no se usa, Python guarda los datos de la matriz en un formato predefinido. Una presentación alternativa del formato, si este es igual para las tres columnas, puede escribirse del siguiente modo:

```
np.savetxt(fout, data_out,fmt="%5.2f")
```

Lo descrito resulta en el mismo formato para el archivo.

Los archivos de entrada y de salida se muestra a continuación:

```
Entrada
  3.6809  -1.6855
  3.4113  -2.5989
...
  6.6905  -3.0042
Salida
  3.6809  -1.6855  -6.2042
  3.4113  -2.5989  -8.8656
...
  6.6905  -3.0042 -20.0996
```

3. I/O veloz en Python

En computación, los cálculos y operaciones matemáticas están optimizados dentro de los programas, dado que en muchos casos el tiempo computacional se gasta en procesos I/O; a saber, leer y guardar archivos. Para realizar los procesos I/O con mayor rapidez, en muchos casos, es mejor guardar los archivos en formato binario, en lugar de guardarlos como archivos de texto plano.

Supongamos que se tiene una colección de datos en una matriz grande (por ejemplo, con un millón de puntos), y se desea guardarla. Una forma de hacerlo

es con los códigos hasta ahora descritos, o utilizando otro tipo de formatos binario, como muestra el programa *testio.py*. El programa, además, vuelve a leer los archivos, e imprime el tiempo que demora realizando dicha operación.

testio.py

```
import numpy as np
import time

# tamaño de matriz
m= 100000
n= 10
a = np.ones((m,n))*1.1

# Guardar datos
f1 = 'dout/fout_testio.bin'
f2 = "dout/fout_testio.npy"
f3 = "dout/fout_testio.dat"
print('Tiempo requerido para guardar')

# formato binario
start = time.time()
a.tofile(f1)
print ('Binario: %8.5f segundos.' %(time.time()-start))

# formato Numpy nativo (binario)
start = time.time()
np.save(f2, a)
print ('Numpy:  %8.5f segundos.' %(time.time()-start))

# formato plano de texto
start = time.time()
```

```
np.savetxt(f3, a)
print ('text:   %8.5f segundos.' %(time.time()-start))

#---------------------------------------------
# Cargar datos
print('')
print('Tiempo requerido para cargar')

# formato binario
start = time.time()
b1 = np.fromfile(f1,dtype='float')
b1 = b1.reshape(m,n)
print ('Binario: %8.5f segundos.' %(time.time()-start))

# formato Numpy nativo (binario)
start = time.time()
b2 = np.load(f2)
print ('Numpy:  %8.5f segundos.' %(time.time()-start))

# formato plano de texto
start = time.time()
b3 = np.loadtxt(f3)
print ('text:   %8.5f segundos.' %(time.time()-start))
```

En un computador Mac BookPro (procesador de 2GHz), el proceso tarda para una matriz de *(100000,10)*:

```
Tiempo requerido para guardar
Binario:  0.03887 segundos.
Numpy:   0.01970 segundos.
text:    1.45722 segundos.
```

```
Tiempo requerido para cargar
Binario:  0.00194 segundos.
Numpy:   0.00249 segundos.
text:    2.20922 segundos.
```

Observemos cómo al usar archivos de texto, la lectura puede ser muy demorada. Así, el aumento de velocidad es de 55 veces, comparando el formato binario al ASCII. Los archivos, además, ocupan espacios de memoria del computador muy distintos:

```
7.6M  fout_testio.bin
7.6M  fout_testio.npy
24M   fout_testio.dat
```

El archivo en el formato nativo *NumPy* con extensión *.npy* ocupa el mismo espacio que uno binario normal, pero el archivo binario no *sabe* que la variable guardada *a* es una matriz, sino que guarda la matriz plana, sin tamaños predefinidos, y es necesario reorganizarla.

En algunos casos, el usuario puede querer guardar múltiples variables o arreglos en un solo archivo, y aún así mantener la información de las dimensiones de los arreglos; algo muy similar a guardar archivos *.mat* en Matlab. Esto se puede hacer con el siguiente comando:

```
np.savez(outfile, x, y)
```

Así, se guardan los arreglos *x* y *y*, los cuales después se pueden cargar con:

```
npzfile = np.load(outfile)
```

A su vez, el ejemplo *testio2.py* muestra cómo guardar un archivo binario en formato *.npz*, y cómo volver a cargarlo.

testio2.py

```
import numpy as np

m= 100000
n= 10
a = np.ones((m,n))*1.1
y = np.random.rand(n)
x = np.random.rand(m)
z = 1.5

print('Variables guardadas')
print('Shape de a, x, y, size(z)')
print(np.shape(a),np.shape(x),np.shape(y),np.size(z))
print('')

# Guardar en formato npz numpy
np.savez("fout_testio2b.npz", a,x,y,m,n,z)
np.savez("fout_testio2.npz", a=a,x=x,y=y,m=m,n=n,z=z)

# Cargar archivos, confirmar tamaños
npfile = np.load("fout_testio2.npz")
n  = npfile.f.n
m  = npfile.f.m
x0 = npfile.f.x
y0 = npfile.f.y
a0 = npfile.f.a
z0 = npfile.f.z
print('Variables cargadas')
print('Shape de a, x, y, size(z)')
print(np.shape(a0),np.shape(x0),
     np.shape(y0),np.size(z0))
```

```
print('')

npfile2 = np.load("fout_testio2b.npz")
n1  = npfile2.f.arr_3
m1  = npfile2.f.arr_4
x1  = npfile2.f.arr_1
y1  = npfile2.f.arr_2
a1  = npfile2.f.arr_0
z1  = npfile2.f.arr_5
print('Variables cargadas 2')
print(npfile2.files)
print('Shape de a, x, y, size(z)')
print(np.shape(a1),np.shape(x1),
     np.shape(y1),np.size(z1))
```

Lo anterior tiene como resultado:

```
Variables guardadas
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1

Variables cargadas
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1

Variables cargadas 2
['arr_0', 'arr_1', 'arr_2', 'arr_3', 'arr_4', 'arr_5']
Shape de a, x, y, size(z)
(100000, 10) (100000,) (10,) 1
```

Es importante anotar que se puede guardar un número indefinido de arreglos o variables. Sin embargo, cuando carga el archivo, a cada variable o vector, *Numpy* le asigna un nombre, a saber:

```
['arr_1', 'arr_0', 'arr_3', 'arr_2', 'arr_4']
```

Sin embargo, es una buena idea guardar las variables con un nombre que pueda ser recordado al cargar el archivo de nuevo, así:

```
np.savez(fname, a=a,x=x,y=y,m=m,n=n)
```

En este último caso, cada arreglo se puede cargar de este modo:

```
m  = npzfile.f.m
x  = npzfile.f.x
```

4. Archivos de texto plano o binarios

Como se ve en este capítulo, hay un *tradeoff* (compensación o sacrificio) entre trabajar con archivos de texto plano o binarios. Los archivos de texto son mas fáciles de manipular; se pueden cargar en Excel o Matlab con facilidad, e incluso se puede abrir el archivo y mirar los datos. Este es entonces el formato sugerido cuando el tamaño del archivo o la velocidad de los cálculos no son un problema.

Sin embargo, para bases de datos muy grandes y procesamiento de datos pesados, es mejor usar formatos binarios, porque permiten transferencia e I/O rápidos, y ocupan menos memoria en el computador. De hecho, Excel muchas veces deja de funcionar cuando tiene que procesar bases de datos *no tan* grandes. Además, en el archivo binario no se tiene que decidir de antemano el formato (y la precisión con la que se quiere guardar los números; *¿cuántos decimales quiero guardar?*). Sin embargo, los archivos binarios son menos portátiles (pasarlos de un Mac a un Windows puede causar problemas), y se requiere conocer en muchos casos el formato del archivo para poder leerlo.

Problemas

1. Escriba un programa que lea un archivo de texto plano que contenga cinco números por línea. Calcule el promedio de los cinco números, y luego sustraiga el promedio de cada uno de los cinco números originales. Guarde un nuevo archivo con los valores con el promedio removido (*demeaned*). Permita que el usuario pueda especificar el nombre del archivo de entrada y salida.

   Por ejemplo, partamos de un caso en el que el archivo de entrada es el siguiente:

   ```
   10 20 30 40 50
   1 2 3 4 5
   2 4 6 8 10
   5 5 5 5 5
   ```

   El resultado de su programa deberá escribir un nuevo archivo, así:

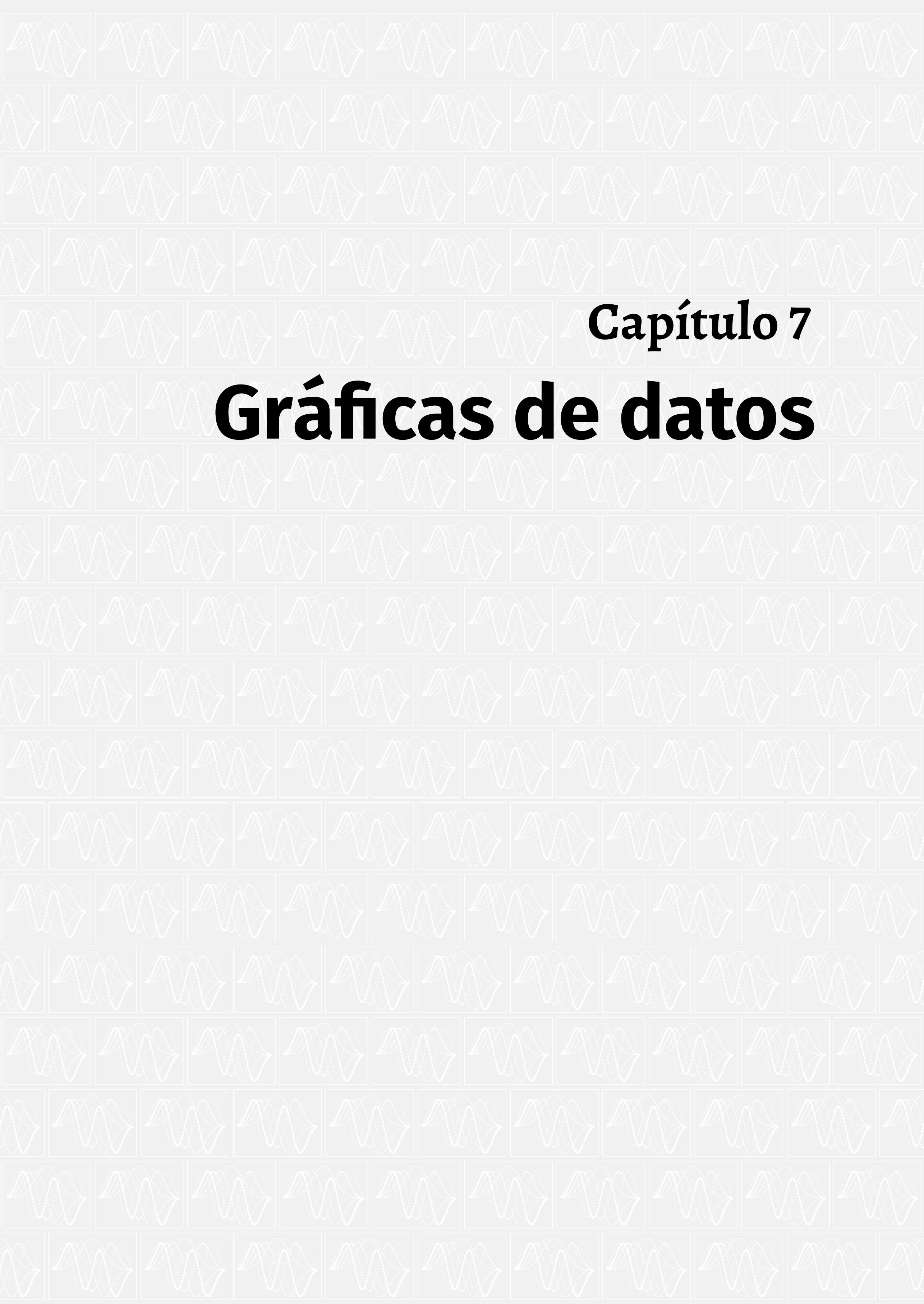
   ```
   -20.00 -10.00  0.00 10.00 20.00
   -2.00 -1.00  0.00  1.00  2.00
   -4.00 -2.00  0.00  2.00  4.00
    0.00  0.00  0.00  0.00  0.00
   ```

2. En geociencias, hay múltiples fuentes de información (geofísica, geología, fallas, geoquímica, etc.). Una de ellas se llama GEOROC, y contiene información sobre geoquímica de rocas en el mundo. Descargue el archivo (link) llamado *ANDEAN_ARC_part1.csv*, que contiene una gran cantidad de datos, incluyendo latitud y longitud de muestreo y contenido en peso de SiO_2 (entre otras muchas variables).

   Utilizando su método preferido (por ejemplo con *Pandas* o *NumPy*), lea el archivo e imprima las primeras 20 líneas con la latitud, longitud y porcentaje de SiO_2 en peso, cuyos valores encuentra en las columnas 4, 6, y 27 respectivamente.

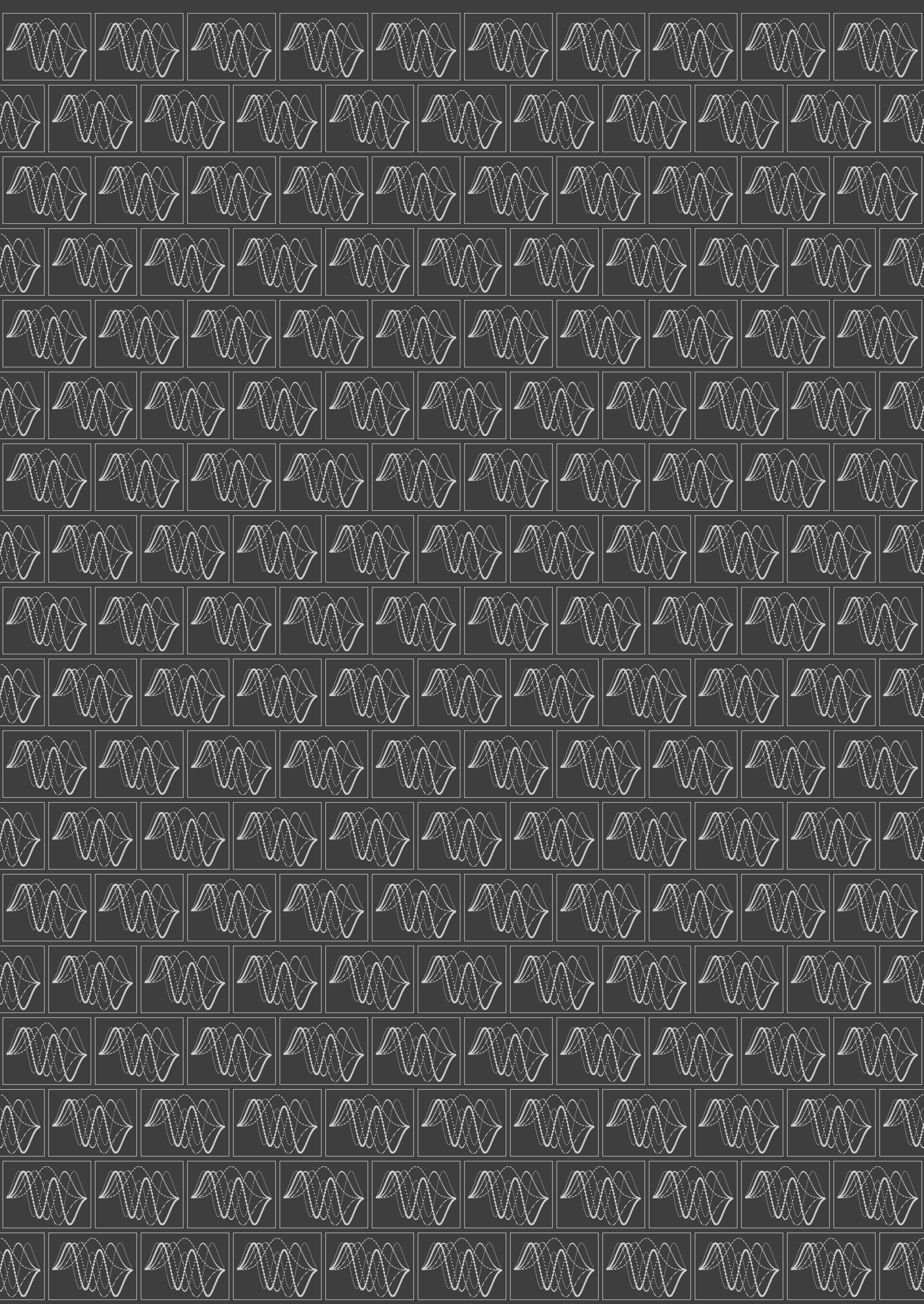
Tenga en cuenta que algunas variables ostentan valores inexistentes así que, dependiendo del método, debe ajustar dichos casos.

3. Usando el ejercicio anterior, describa estadísticamente las características de SiO_2, Al_2O_3, FeO_{total}, CaO, MgO (columnas 27, 30,34,35,36). Tenga en cuenta que en algunos casos hay espacios vacíos, que no deben ser tenidos en cuenta en el cálculo.



Capítulo 7

Gráficas de datos



Una de las grandes ventajas de aprender programación en Python, a diferencia de Fortran o C/C++, es que Python tiene paquetes para generar figuras de alta calidad (aunque la recomendación del autor es aprender igualmente Fortran o C/C++). El poder de los paquetes de gráficas en Python es mucho mayor que lo que se puede presentar en este libro y se recomienda continuar explorando el potencial de los paquetes que acá se discuten. Además, también se incluye una introducción a dos paquetes de generación de mapas de alta calidad (*Cartopy* y *PyGMT*). Ambos paquetes deben estar instalados para que Python los pueda importar.

Antes de empezar, es fundamental que una figura tenga toda la información necesaria para que cualquier persona que la vea la pueda entender. Por eso, cada eje debe estar descrito, y si se tienen más de un tipo de datos, se debe explicar qué significa cada uno. Para la presentación de resultados científicos, especialmente en geociencias, es importante generar figuras que muestren la información de manera correcta, veraz y con buena resolución. No es aceptable entregar figuras sin ejes explicados o escalas, si es necesario.

Como comentario personal, aunque se muestran algunos ejemplos de figuras en 3D, estas en muchos casos (aunque atrayentes estéticamente) no son útiles para presentar la información. Las páginas de revistas o la pantalla del computador son 2D, de modo que no es fácil percibir la tercera dimensión. A veces, es mejor un mapa de contornos, que mostrar la topografía en 3D en una figura.

En este contexto, el paquete que se utilizará es *Matplotlib*, y, en concreto, *pyplot*, que por lo general hacen parte de la instalación básica de Anaconda. Asimismo, los paquetes de mapas deben instalarse, por ejemplo, con ayuda del archivo *geopython.yml* incluido en el Capítulo 1.

1. Concepto de gráficas orientada a objetos

El primer ejemplo de generación de una figura usando *Matplotlib* permite graficar *(x,y)*. Sin embargo, esta **no es la mejor forma de sacar provecho** del paquete, y, al crear figuras más complejas, puede dar lugar a confusiones.

El programa *naive_plot.py* produce una figura de la magnitud de un terremoto versus la caída del esfuerzo, con el comando *plt.semilogy* de los datos de [25]. **Este tipo de programas para generar gráficas no se recomienda**.

naive_plot.py

```
import numpy as np
import matplotlib.pyplot as plt
# cargue los datos de Poli
fname = 'data/poli_2016.dat'
data = np.loadtxt(fname,usecols=[4,5,6])

plt.semilogy(data[:,0],data[:,1],'o')
plt.show()
```

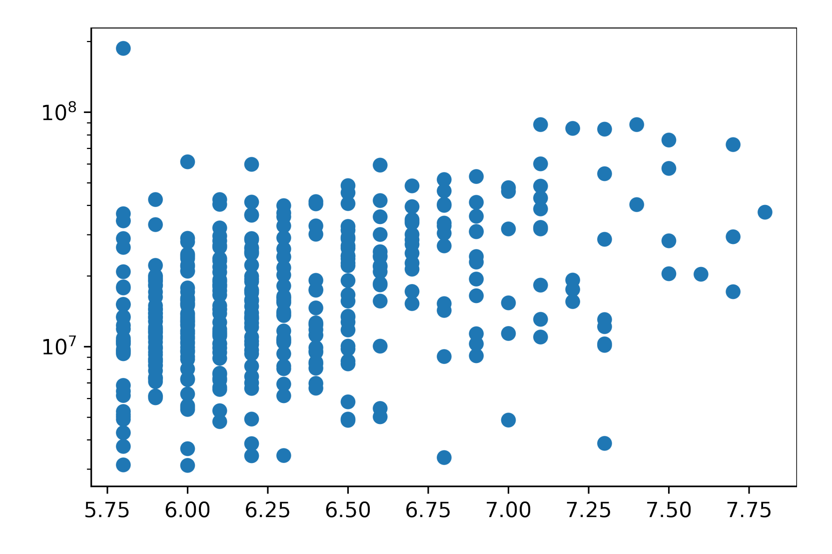


Figura 7.1. Resultado de correr *naive_plot.py*.

El paquete *matplotlib* usa el concepto de programación orientada a objetos (object-oriented); por consiguiente, es importante, antes de empezar a hacer figuras, entender los dos objetos principales o más comunes que se usan en *matplotlib*:

- *figure*, es el objeto que corresponde al contenedor principal de todos los elementos de la imagen.
- *axes*, es el objeto que representa la región de la imagen para desplegar los datos.

En otras palabras, *figure* es el tablero completo (*canvas*) y *axes* es el espacio para graficar los datos (puede pensarse como los *ejes*). Por ejemplo, este es el procedimiento para preparar una figura con un único panel o *axes*:

first_plot.py

```
import numpy as np
import matplotlib.pyplot as plt

...

fig = plt.figure()
ax  = fig.add_subplot(111)
ax.semilogy(data[:,0],data[:,1],'o')
plt.show()
```

Un resultado idéntico se obtiene con lo que sigue:

```
fig = plt.figure()
ax  = fig.add_subplot(111)
```

De esta manera, se tiene completo control de los dos objetos *figure* y *axes*, y es claro que cualquier comando será aplicado al objeto *ax*, que pertenece a *fig*. Además, a menos que se especifique con un nuevo *plt.subplots*, la figura activa (*current figure*) será la variable *fig*. Así, se pueden crear tantos *axes* o subplots en una sola figura como se requiera. Para simplificar las cosas, los términos *axes* y subplots refieren a lo mismo de ahora en adelante, y son tratados como sinónimos.

Con el objetivo de generar una figura con varios subplots, lo cual puede ser común en el trabajo de investigación, veamos el siguiente ejemplo:

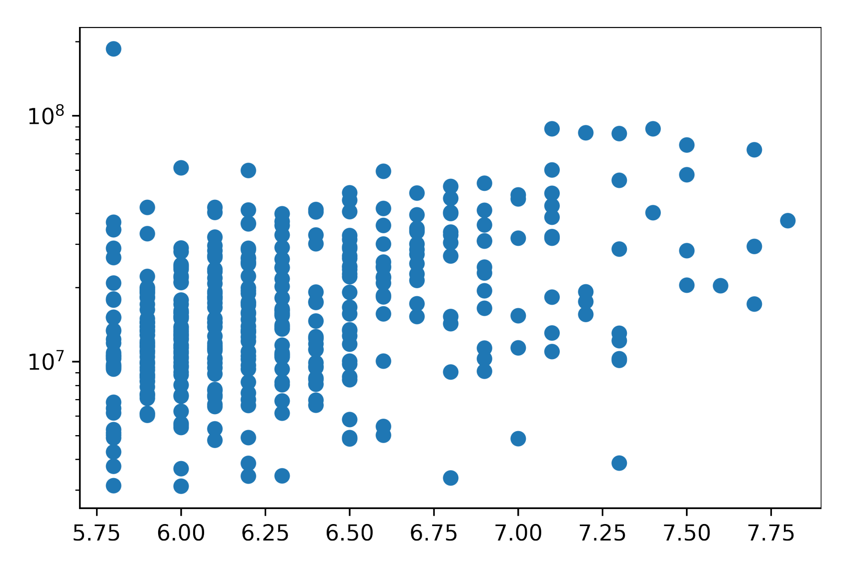


Figura 7.2. Resultado de correr *first_plot.py*, con idéntico producto a la figura 7.1.

two_subplots.py

```
import numpy as np
import matplotlib.pyplot as plt

fig,ax = plt.subplots(nrows=1,ncols=2)
print(ax)
```

En este marco, la variable *ax* es un arreglo con dos subplots:

```
[<AxesSubplot:> <AxesSubplot:>]
```

El programa genera una figura *fig* y dos *subplots*, en este caso, *ax[0]* y *ax[1]*, que están organizados en una fila y dos columnas. Una alternativa para separar los dos *subplots* es:

```
fig,(ax1,ax2) = plt.subplots(nrows=1,ncols=2)
```

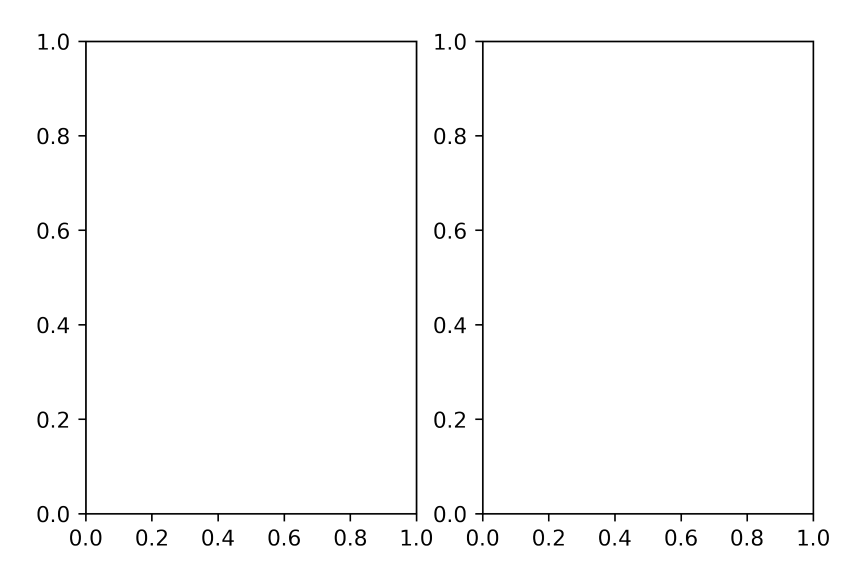


Figura 7.3. Resultado de correr *two_subplots.py*, donde se crean dos *subplots* vacíos.

El comando *subplots()* permite especificar una grilla para ubicar *subplots*. En este caso, los parámetros *nrows* y *ncols* se usan para definir el número de filas y columnas de la grilla que se quiere utilizar para ubicar cada uno de los *subplots*. A continuación, un ejemplo más complejo:

four_subplots.py

```
fig, ((ax1,ax2), (ax3,ax4)) = plt.subplots(2,2,
                              figsize=(9,5))
ax2.semilogy(data[:,0],data[:,1],'ko')
ax3.semilogy(data[:,0],data[:,2],'r^')
plt.show()
```

De tal modo, se muestra también que se puede definir el tamaño de la figura (*fig*) con el parámetro *figsize*. Es importante anotar que los subplots *ax2* (negro) y *ax3* (rojo) tienen datos.

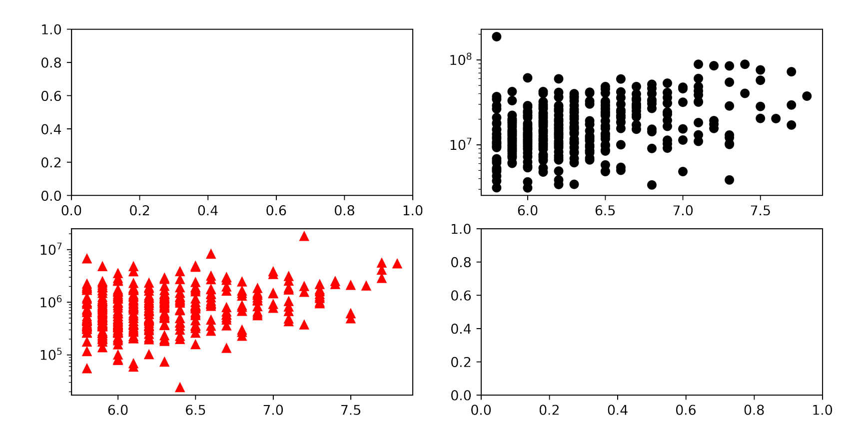


Figura 7.4. Resultado de correr *four_subplots.py*. Se crean cuatro *subplots*.

2. Gráficas 1D/2D

En muchas ocasiones, para una publicación, por ejemplo, se requiere comparar varias curvas o series de datos. Esto se puede hacer en una sola figura, mostrando cada curva con diferentes características (líneas continuas, punteadas, de puntos, con diferentes símbolos, etc.).

En el programa *plot_lines.py*, se busca desplegar en una figura varios arreglos de datos. Dicho programa muestra la forma de generar diferentes tipos de líneas, la leyenda, y cómo solicitar límites mínimos y máximos en los ejes. Por último, cómo se puede guardar la figura en un archivo digital, que en este caso es un *.png*. Otros formatos incluyen *.ps*, *.eps*, *.pdf*, *.svg*, etc. En los siguientes ejemplos, los comandos *plt.savefig* y *plt.show()* no se muestran para evitar ocupar demasiado espacio.

El programa *plot_lines.py* carga un arreglo con las funciones Slepian (usadas en análisis de series de tiempo y métodos de Fourier) y sus valores propios [36, 28]), disponibles en el archivo *slepian.npz*. Los objetivos son:

- Graficar las cinco curvas de la variable *dpss*, usando diferentes colores y/o formatos.
- Nombrar la figura y los ejes.

- Poner la leyenda, usando los valores propios.
- Poner límites en el eje *y*, para que sea simétrica.
- Guardar la figura en algún formato (PDF, PNG, etc.).

plot_lines.py

```
import numpy as np
import matplotlib.pyplot as plt

fdat = np.load('data/slepian.npz')
x    = fdat.f.x
dpss = fdat.f.dpss
v    = fdat.f.v

fig,ax = plt.subplots(figsize=(8,7))
ax.plot(x,dpss[:,0],marker='^',linestyle=' ',label=v[0])
ax.plot(x,dpss[:,1],":",label=v[1])
ax.plot(x,dpss[:,2],"--",label=v[2])
ax.plot(x,dpss[:,3],marker='.',label=v[3])
ax.plot(x,dpss[:,4],'-s',label=v[4])

ax.set_xlabel('tiempo')
ax.set_ylabel('Amplitud')
ax.set_title('Funciones Slepian y su concentración')
ax.set_ylim((-0.22, 0.22))
ax.legend()

plt.savefig('outfig.pdf')
plt.show()
```

El resultado se muestra en la figura 7.5.

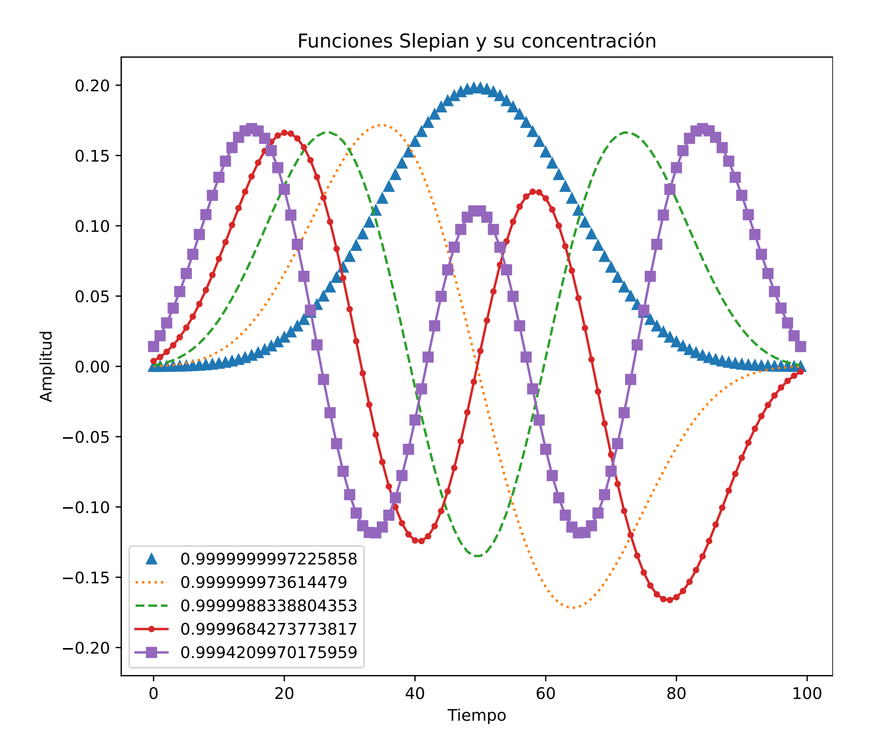


Figura 7.5. Resultado de correr *plot_lines.py*.

Por su formato, las líneas en la figura pueden desplegarse como continuas, punteadas, etc., o ser reemplazadas con símbolos. El formato de la primera columna de datos se solicita así:

```
ax.plot(x,dpss[:,0],marker='^',linestyle=' ')
```

El comando anterior grafica triángulos sin una linea que las una, mientras que la segunda columna se solicita así:

```
ax.plot(x,dpss[:,1],':')
```

Acá, se obtiene una línea punteada (no continua). Los colores en este caso son automáticamente seleccionados por Python, que tiene una secuencia de colores especificada (azul, amarillo, verde, rojo, etc.). En tanto, los formatos de las líneas y de los *markers* son muy variados, y están por fuera del objetivo

de este libro. Una explicación más precisa se puede encontrar en la documentación de *matplotlib*. Además, el grosor de la línea, el relleno de los símbolos, etc., pueden ser modificados.

También es importante destacar que el comando *ax.plot(x,y)* genera la figura con escala lineal-lineal, aunque también se puede usar *semilogx*, *semilogy* o *loglog*, con escala logarítmica con base 10.

Se puede adicionar texto a cada eje, y título de la figura, como se muestra a continuación:

```
ax.set_xlabel('tiempo')
ax.set_ylabel('Amplitud')
ax.set_title('Funciones Slepian y su concentración')
```

Asimismo, leyenda de los símbolos:

```
ax.set_ylim((-0.22, 0.22))
ax.legend()
```

Además, se puede poner límites de los ejes en la gráfica (en *x* o *y*) y guardar la figura:

```
ax.set_ylim((-0.22, 0.22))
plt.savefig(fname)
```

Acá, el formato para guardar la figura puede ser *.ps, .eps, .png, .svg*, etc. Incluso la leyenda puede tener un formato más amigable:

```
ax.plot(x,dpss[:,1],':',
    label=r'$\lambda_1$=%12.10f '%(v[1]))
```

De este modo, la variable *v[1]* tiene un formato específico (con diez decimales y doce espacios en total).

En Notebooks, no es necesario usar un comando para que la figura sea desplegada en la pantalla, pero si se quiere desplegar la figura corriendo el programa en Python, se debe pedir con el siguiente comando:

```
plt.show()
```

2.0.1. Figuras con barras de error

En casi todos los casos en investigación en geociencias (o en ciencias en general), es necesario presentar intervalos de confianza, con el fin de poder obtener conclusiones tan robustas como sea posible. ¿Cómo? Presentando, no un valor único, sino una gama de posibles valores (intervalos, regiones, etc.), de tal manera que haya un alto nivel de confianza de que el valor *verdadero* esta dentro de todos ellos [37].

El archivo *source_error.dat* muestra los valores estimados y su intervalo de confianza (5-95 %) del momento sísmico (M_0), la frecuencia de esquina (f_c) y la caída del esfuerzo (τ), para una serie de terremotos [30].

source_error.dat

```
ID  ML   M0(Nm)      M0_conf         fc(Hz)     fc_conf    ...
01 1.1  0.96e11 1.90e10 2.10e10    43.80  10.50 13.83 ...
02 3.5  1.73e14 6.30e13 9.40e13     6.92   2.15  3.11 ...
...
```

En tanto, el código *plot_errorbar.py* lee el archivo y genera una figura con dos *subplots* de M_0 vs. f_c en el primer panel y de M_0 vs. τ en el segundo, incluyendo sus barras de error. Tengamos en cuenta que:

- Los intervalos de confianza representan la longitud del error, no el valor correspondiente a la posición de la barra de error.
- El código cambia la escala a logarítmica en ambos ejes.
- El código marca los ejes e incluye una grilla.

A su vez, el código *plot_errorbar.py* muestra otra forma de trabajar con los módulos de gráficas de Python. Observemos que primero se crea una figura con subpaneles *plt.subplots*, donde se genera una variable *axs* que se usa para

producir las figuras de cada subpanel, usando *ax1 = axs[0]* para el primer panel, y *ax2 = axs[1]* para el segundo panel.

plot_errorbar.py

```
import numpy as np
import matplotlib.pyplot as plt

# Cargar datos
fname = "data/source_error.dat"
data = np.loadtxt(fname,skiprows=1)
M0       = data[:,2]
M0_err   = data[:,3:5].T
fc       = data[:,5]
fc_err   = data[:,6:8].T
tau      = data[:,8]
tau_err  = data[:,9:11].T

# La figura
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8,6))
ax1.errorbar(M0, fc,xerr=M0_err, yerr=fc_err,fmt='o')
ax1.set_ylim([1 , 100])
ax1.set_xlim(1e10, 1e15)
ax1.set_yscale('log')
ax1.set_xscale('log')
ax1.grid(which='both')
ax1.xaxis.tick_top()
ax1.xaxis.set_label_position("top")
ax1.set_xlabel('Seismic Moment (N.m)');
ax1.set_ylabel('Corner frequency (Hz)')

ax2.errorbar(M0, tau,xerr=M0_err, yerr=tau_err,
    fmt='^',color='black', ecolor='lightgray')
```

```
ax2.set_yscale('log')
ax2.set_xscale('log')
ax2.set_ylim([0.5 , 200])
ax2.set_xlim(1e10, 1e15)
ax2.set_xlabel('Seismic Moment (N.m)');
ax2.set_ylabel('Stress Drop (MPa)');
```

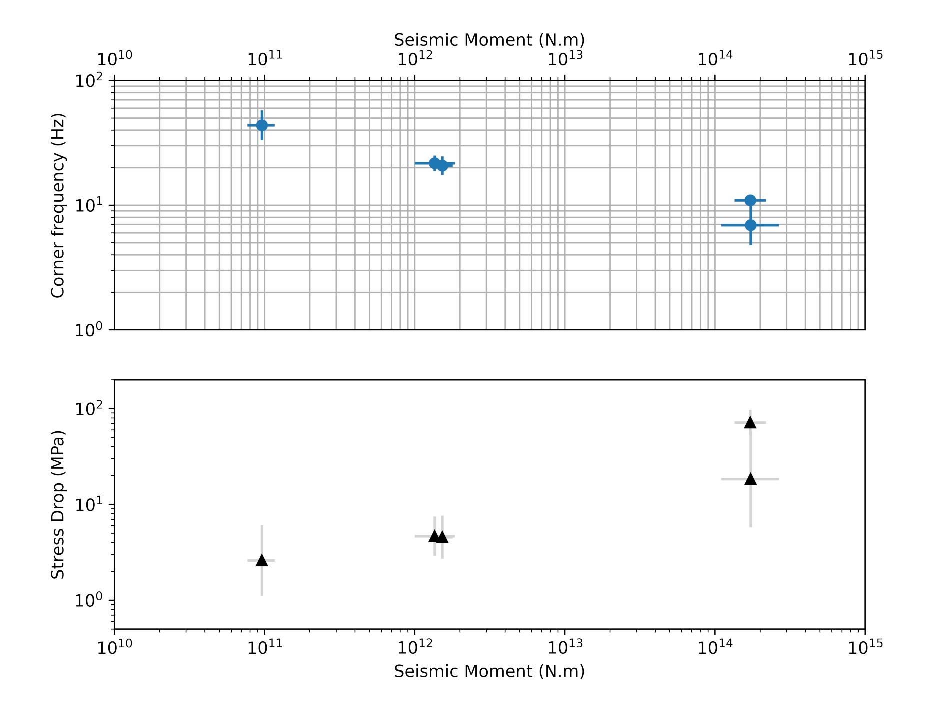


Figura 7.6. Resultado de correr *plot_errorbar.py*.

Una vez cargados los datos, se generan la figura y sus dos *subplots* en una sola columna y dos filas:

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8,6))
```

Matplotlib tiene un comando para generar figuras de barras de error.

```
ax1.errorbar(M0, fc,xerr=M0_err, yerr=fc_err,fmt='o')
```

En este caso, se gráfica M_0 en el eje x y f_c en el y, con sus respectivas barras de error con *xerr* y *yerr*. Al usar *fmt='o'*, se grafican las barras de error y el valor con un símbolo, y no hay líneas que unan los puntos.

Cabe señalar que *errorbar()* espera que el usuario provea la longitud de la barra de error, desde el valor central hasta el limite superior e inferior, no el valor máximo o mínimo. Por ejemplo, según la tabla para la variable f_c, se tienen los siguientes parámetros:

```
fc = 43.80   fc_conf 10.50 13.83
```

Esto, en realidad, corresponde a que el intervalo de confianza sería:

```
fc = 43.80   (33.30 - 57.63)
```

Asimismo, en el ejemplo, las barras de error son asimétricas, dado que los arreglos *xerr* tienen dimensiones *(2,N)* para describir la extensión de la barra de error con los límites inferior y superior. Si la barra es simétrica, *xerr* puede ser un arreglo *(N,)*, y si todas las barras son de igual longitud para todos los puntos, *xerr* puede ser un número.

El programa además le solicita a Python que la gráfica sea elaborada en escala logarítmica en ambos ejes:

```
ax1.set_yscale('log')
ax1.set_xscale('log')
```

Se puede agregar la grilla de este modo:

```
ax1.grid(which='both')
```

Y, para mejorar la presentación de la figura moviendo la descripción del eje a la parte superior, se usa el siguiente procedimiento:

```
ax1.xaxis.tick_top()
ax1.xaxis.set_label_position("top")
```

De forma similar se puede mover al lado derecho para el eje *y*, con los siguientes comandos:

```
ax1.yaxis.tick_right
ax1.yaxis.set_label_position("right")
```

2.0.2. Histogramas

En algunos casos, es necesario mostrar la distribución de una serie de datos, para estudiar si estos tienen una distribución normal (gaussiana), o de otro tipo. El programa *plot_histogram.py* genera un histograma a partir de datos de caída del esfuerzo y esfuerzo aparente de más de 400 terremotos profundos a nivel mundial [25]. Los datos están en el archivo *poli_2016.dat*.

plot_histogram.py

```
import matplotlib.pyplot as plt
import numpy as np

# cargar datos Poli

fname = 'data/poli_2016.dat'
data  = np.loadtxt(fname,usecols=[5,6])

# crear bins y labels eje x
bins=np.linspace(5,8.5,20)
labels = [0.1, 1.0,10, 100]

# figura
fig, ax = plt.subplots()
ax.hist(np.log10(data[:,0]),bins=10,
        label='Stress Drop')
ax.hist(np.log10(data[:,1]),bins=bins,
        label='Apparent Stress')
```

```
ax.set_xlim(5, 8)
ax.set_xticks((5,6,7,8))
ax.set_xticklabels(labels)
ax.set_xlabel('Stress Drop/Apparent Stress (MPa)')
ax.set_ylabel('Count')
ax.set_yticks([])
plt.legend()
```

La figura 7.7 muestra el histograma resultante. En ese sentido, resulta útil el comando:

```
hist(X,bins=10)
```

Este divide el rango de los datos en *x* en 10 bandas, de igual espesor, que Python define de manera automática. Podemos solicitar un espaciamiento diferente con:

```
hist(X,bins=np.linspace(5,8.5,20))
```

Acá, el usuario puede definir los límites de cada banda, en este caso entre 5 y 8.5 con 20 barras de igual espesor. Cabe señalar que se retiraron los marcadores del eje *y* a propósito con el siguiente comando:

```
ax.set_yticks([])
```

3. Gráficas 2D/3D

En geociencias, muchas veces se toman datos en la superficie de la Tierra, y se busca mostrarlos en mapa o en sección cruzada, o en figura 3D. Sin embargo, es importante tener en cuenta que las figuras en 3D muchas veces no son muy útiles para mostrar los datos. En los siguientes ejemplos, se muestran varias formas de presentar unos datos *z* tomados en la posición *x* y *y*, donde *z* puede representar topografía, anomalías geofísicas, contenido de SiO_2, etc.

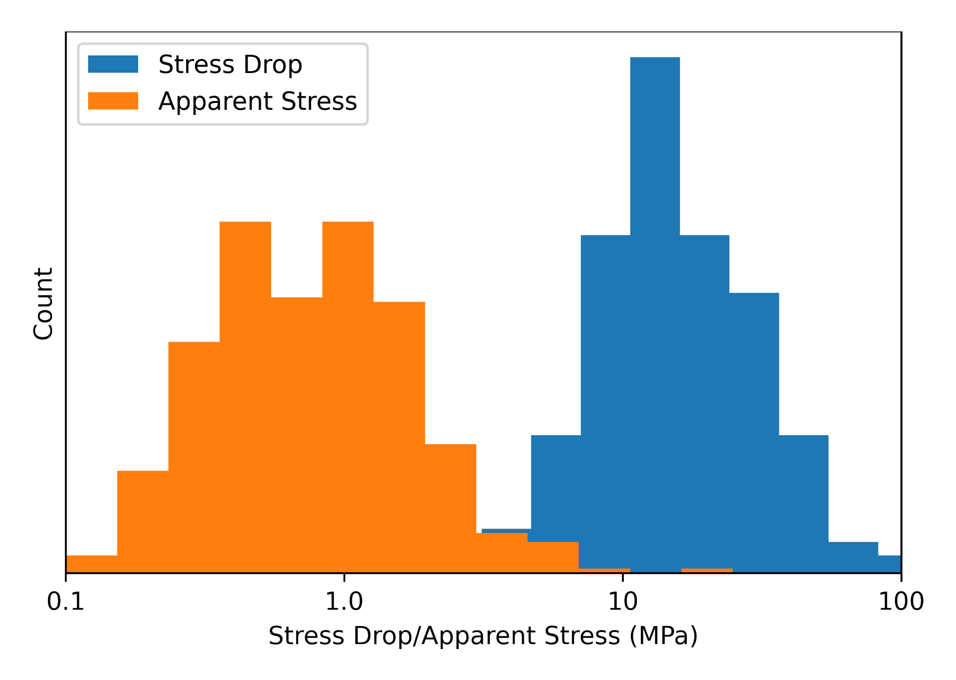


Figura 7.7. Resultado de correr *plot_histogram.py*.

Se proveen dos colecciones de datos sintéticos que representan anomalías gravimétricas en los siguientes archivos:

- *rand_matrix.npz* - Anomalías registradas en 400 puntos aleatorios en posiciones *x* y *y*

- *grid_matrix.npz* - Anomalías registradas en una grilla uniforme (45×45) en *x* y *y*

El programa *plot_scatter.py* carga los datos de anomalías con un muestreo aleatorio en *x* y *y*, algo parecido a la realidad en la que no siempre es posible recoger datos en una grilla homogénea. La figura 7.8 muestra una gráfica tipo *scatter* en la cual las posiciones de los puntos están dadas por *(x,y)*, y el tamaño y color de cada punto dependen del valor de la anomalía *z*. A su vez, la figura 7.9 muestra una gráfica 3D, utilizando el comando *plot_trisurf*, que genera una superficie 3-D triangular, cuyos puntos están definidos por *x*, *y*, *z*.

plot_scatter.py

```
import matplotlib.colors as cm
import matplotlib.pyplot as plt
import numpy as np

fdat = np.load("data/rand_matrix.npz")
x = fdat.f.x
y = fdat.f.y
z = fdat.f.z

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8,8))
im1 = ax1.scatter(x,y,z*40+15,c=z,
    cmap='terrain',edgecolor='gray')
ax1.set_xticks([])
plt.colorbar(im1,ax=ax1)
ax1.set_title('linear color scale')

im2 = ax2.scatter(x,y,z*40+15,c=z,norm=cm.LogNorm(),
    cmap='terrain',edgecolor='k')
ax2.set_title('Log color scale')
plt.colorbar(im2,ax=ax2)

fig2 = plt.figure(figsize=(8,8))
ax = fig2.add_subplot(111, projection='3d')
ax.plot_trisurf(x,y,np.log10(z))
ax.set_title('Tri-Surf log10(Z)')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('log(Z)')
ax.view_init(45, 135)
...
```

Para el primer *subplot* de la figura 7.8, se crea un *scatter*, y se adiciona la barra de colores. El *scatter* es en realidad una forma de mostrar unos datos con tres dimensiones en un plano 2D. Asimismo, Python también da opción de hacer figuras tipo *scatter* en 3D, aunque su uso no resulta recomendable, porque no muestra los datos con facilidad.

```
im1 = ax1.scatter(x,y,z*40+15,c=z,cmap='terrain')
plt.colorbar(im1,ax=ax1)
```

Así, el tercer valor *z*40+2* representa el tamaño del símbolo (en puntos), y *c=z* determina el color del símbolo de acuerdo con la escala de colores definida en *cmap='terrain'*. Además, *Matplotlib* ofrece una gran gama de paletas de colores que se pueden consultar en su página de referencia. Cabe señalar además que posteriormente se define un nuevo objeto *im1= ax1.scatter* el cual es necesario para asignarle el *colorbar* a la gráfica correspondiente.

El segundo *subplot* es idéntico al primero, excepto porque la escala de colores que representa la variable *z* es logarítmica:

```
im2 = ax2.scatter(x,y,z*40+2,c=z,
        norm=cm.LogNorm(), cmap='terrain')
```

Con ello se busca representar los datos de anomalías de otra manera, y, en muchos casos (cuando las anomalías varían en ordenes de magnitud, por ejemplo, con resistividades, magnitudes de terremotos, etc.), esta representación puede ser muy útil.

Adicionalmente, el programa *plot_scatter.py* también genera una figura 3D de los mismos datos (figura 7.9), por medio de la creación de una superficie formada por triángulos.

```
ax = fig.add_subplot(111, projection='3d')
ax.plot_trisurf(x,y,np.log10(z))
```

Es importante tener en cuenta que en este caso es necesario crear el subplot *ax*, y declarar que se solicita una proyección con *projection='3d'*. Luego, con

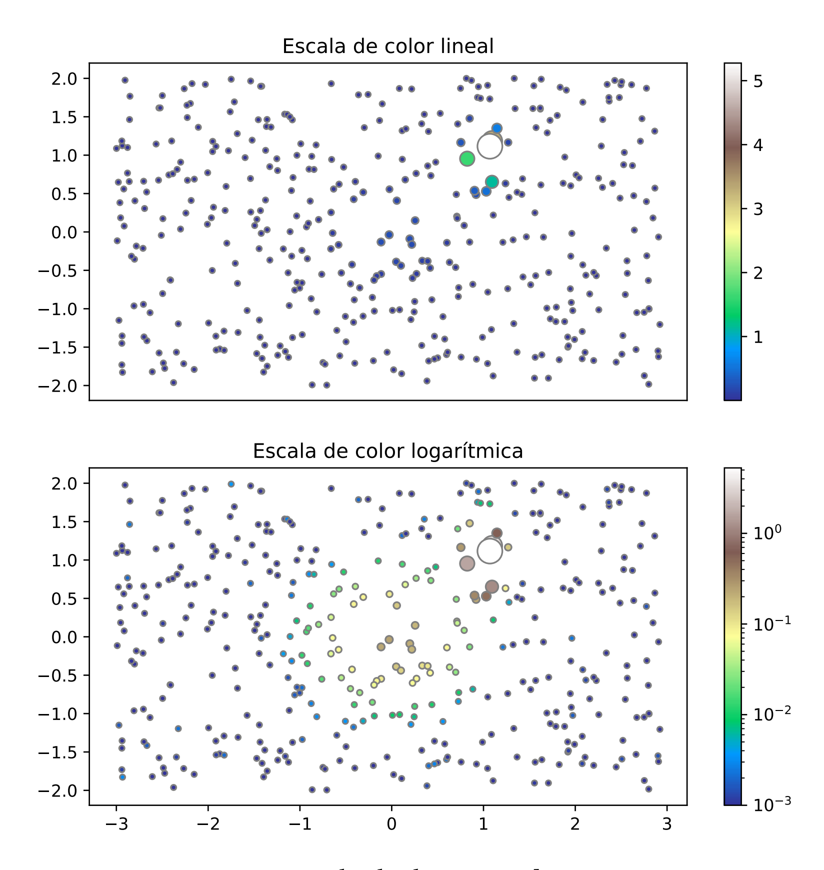


Figura 7.8. Resultado de correr *plot_scatter.py*.

plot_trisurf, se ordena a Python que genere la superficie. En figuras 3D, es posible cambiar el ángulo de vista de la figura con el siguiente comando:

```
ax.view_init(45, 135)
```

Acá, el primer valor representa el ángulo de elevación del observador (en grados) y el segundo representa el azimut (con respecto a *x*, también en grados).

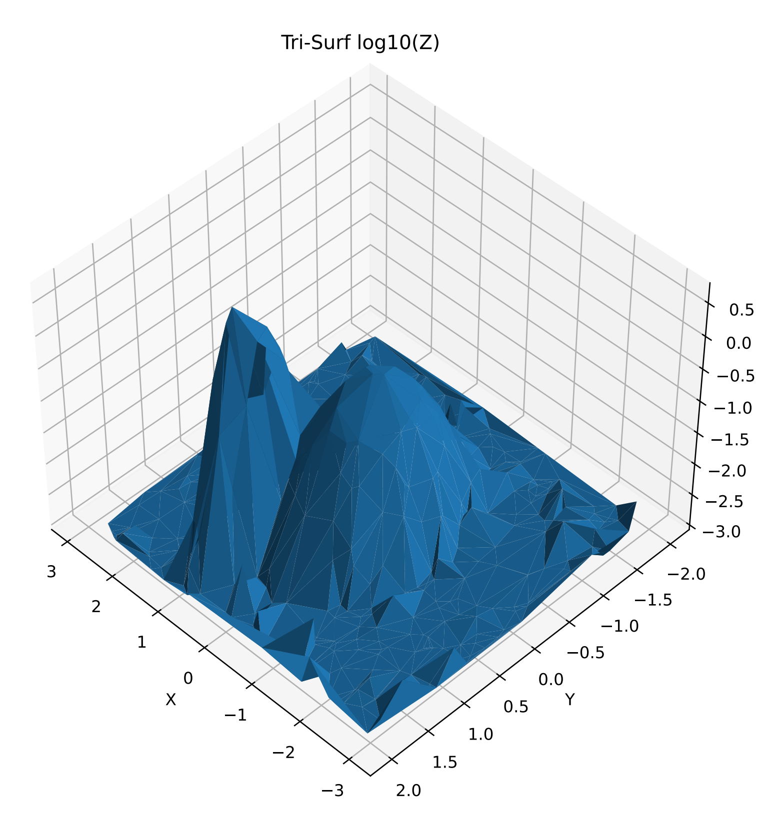


Figura 7.9. Resultado del comando *plot_trisurf* en *plot_scatter.py*.

El programa *plot_contour.py* carga los datos de anomalías con un muestreo en una grilla uniforme en x y y, lo cual permite usar programas que elaboran figuras de contornos. De este modo, la figura 7.10 muestra dos *subplots* con figuras de contornos, uno de ellos con curvas de nivel para la variable z y el otro con contornos rellenos, siguiendo una paleta de colores. Por su parte, la figura 7.11 muestra una gráfica 3D efectuada utilizando el comando *plot_surface*, que genera una superficie 3-D de los datos x, y, z, similar a la figura 7.9.

plot_contour.py

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.colors as cm
import matplotlib.pyplot as plt
```

```
import numpy as np

fdat = np.load("data/grid_matrix.npz")
...

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8,8))
im1 = ax1.contour(x,y,z,[1e-3, 1e-2,1e-1,1e0,1e1],
                  colors='k')
ax1.clabel(im1,inline=1,fontsize=8,fmt="%4.0e")
...
im2 = ax2.contourf(x, y, z,norm=cm.LogNorm(),
        cmap='ocean')
fig.colorbar(im2,ax=ax2)

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(x, y, np.log10(z),cmap='ocean')
ax.view_init(45, 135)
```

Arriba, el comando *contour()* permite construir curvas de nivel de la matriz *z*, además de definir cuáles valores de curvas de nivel desplegar, así:

```
im1 = ax1.contour(x,y,z,[1e-3, 1e-2,1e-1,1e0,1e1],
                  colors='k')
ax1.clabel(im1,inline=1,fontsize=8,fmt="%4.0e")
```

Las curvas de nivel en este caso son negras (*colors='k'*), y se le puede dar formato a la nomenclatura (*fmt=*).

El segundo *subplot* también es una figura de contornos pero, en este caso, usa una paleta de colores:

```
im2 = ax2.contourf(x, y, z,norm=cm.LogNorm(),
        cmap='ocean')
```

```
fig.colorbar(im2,ax=ax2)
```

En este caso, el comando *contourf* no ordena cuáles valores de contornos debe desplegar, sino que Python lo elige de forma automática. Notemos además que acá se utiliza la normalización logarítmica (*norm=cm.LogNorm()*), la cual es similar a la figura de contornos anterior, en la cual el usuario eligió cuáles curvas de nivel desplegar.

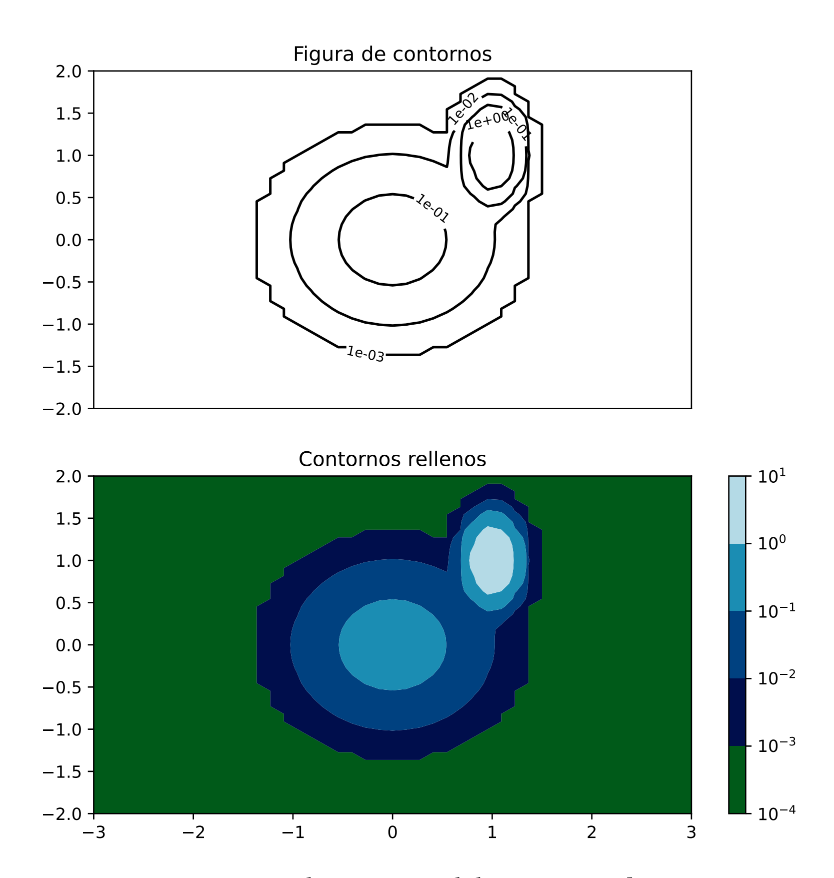


Figura 7.10. Figuras de contorno del programa *plot_contour.py*.

En la siguiente figura (7.11) se muestran los mismos datos, pero ahora en 3D, con colores para representar las alturas, así:

```
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(x, y, np.log10(z),cmap='ocean')
ax.view_init(45, 135)
```

Observemos como el comando *plot_surface* (similar a *surf* en Matlab) sigue el mismo patrón de los códigos anteriores.

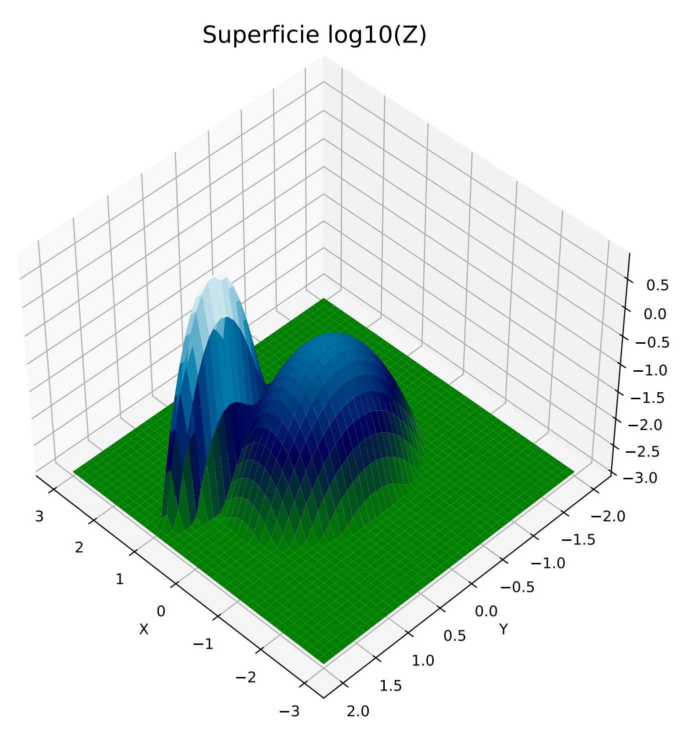


Figura 7.11. Figuras de superficie en 3D del programa *plot_contour.py*.

Para terminar, se recomienda visitar `https://matplotlib.org/`, donde se muestran muchos ejemplos de cómo generar figuras con Python. En este libro no es posible revisar todo lo que quisieramos, pero contamos con que esta introducción le proporcione al lector las herramientas idóneas para continuar su aprendizaje, para generar figuras de calidad.

Problemas

1. Descargar el archivo *chap7_data.dat* y generar archivos con formato *.png*.

   a) Graficar columna 1, contra columnas 2 y 3, en figuras separadas.

   b) Graficar columna 1, contra columnas 2 y 3, en la misma figura.

   c) Graficar columnas 4 contra 5, como nube de puntos, de color rojo, y columnas 4 y 6 como nube de puntos de otro color. Interpretar lo que se observa.

2. Utilizando la ecuación de movimiento parabólico:

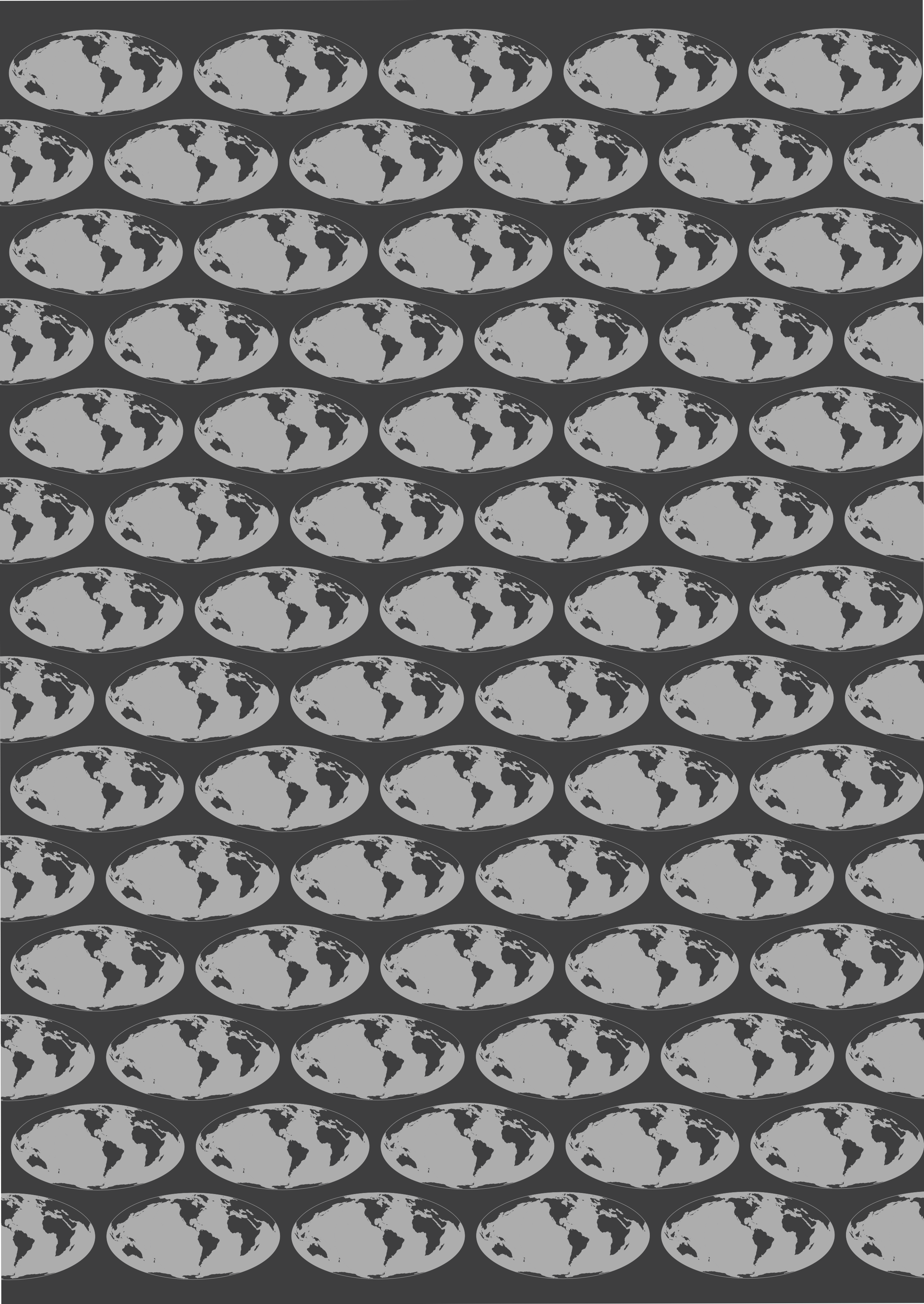
$$
\begin{aligned}
x &= v_0 t cos(\theta), \\
y &= v_0 t sin(\theta) - \frac{1}{2} g t^2,
\end{aligned}
$$

   donde t es el tiempo, v_0 es la velocidad inicial, g es la aceleración de la gravedad (9.8 m/s^2), (x, y), la posición en el tiempo t, y θ es el ángulo de disparo con respecto a la horizontal.

   Genere dos figuras de dos disparos con diferentes ángulos, mostrando mediante una línea punteada la trayectoria del movimiento parabólico de cada uno y, en forma de puntos (o triángulos), la posición (x, y) cada tiempo t (por ejemplo, cada segundo). Es decir, no se deben graficar *todos* los puntos, solo cada cinco o diez puntos.

Capítulo 8

Mapas con *Cartopy* y *PyGMT*



A pesar de que el objetivo de este libro no es involucrar paquetes específicos de geociencias, debe señalarse que la generación de mapas de alta calidad es fundamental para la publicación de trabajos de investigación en la disciplina. Por lo tanto, acá se incluye una introducción a dos paquetes de generación de mapas (*Cartopy* y *PyGMT*). Ambos paquetes deben estar instalados para que Python los pueda importar.

Estos dos paquetes están en creación, con versiones *v0.19* y *v0.4.0*. El hecho de que las versiones para ambos paquetes empiecen con *0.x.y*, significa que los paquetes están en etapa de desarrollo. Esto también significa que los comandos pueden cambiar en cualquier momento, y no siempre hay compatibilidad con versiones anteriores.

PyGMT es un paquete que utiliza el famoso programa *GMT* o Generic Mapping Tools, cuya primera versión fue creada en 1988 [42, 40]. *GMT* es un programa (versión actual 6.0) de líneas de comando que genera figuras *postscript* o *PS*. Estas figuras se pueden convertir a PDF sin problemas. Además, *GMT* tiene varias funciones para generar figuras en el plano cartesiano, mapas, filtros, y mucho más que no se discutirá en este libro. Es posible también que muchas de las figuras en artículos publicados en revistas científicas en geociencias usen *GMT*.

Por su parte, *Cartopy* es un paquete para procesamiento de datos geoespaciales y generación de mapas [24]. La gran diferencia con programas similares es que *Cartopy* usa *Matplotlib* para generar los mapas, y, por lo tanto, tiene todas las ventajas interactivas de *Matplotlib*, y su *lenguaje* sigue las reglas que se discutieron en capítulos anteriores.

En esta sección, se mostrará cómo generar mapas de alta calidad con ambos paquetes. No se hace entonces una compilación exhaustiva de todas las capacidades de ambos paquetes, solo una introducción.

Hay muchas cosas que uno quiere hacer cuando busca desplegar información en mapas. En los ejemplos a continuación, usamos *Shapefiles* y *Grids* disponibles, de modo que no se discute cómo incluir información que el usuario pueda tener, como, por ejemplo, *shapefiles* de carreteras o caminos, o grillas (por ejemplo, en formato *NetCDF*) con información que el usuario haya tomado en campo (gravimetría, sísmica, etc.). En ambos paquetes, es posible des-

plegar esta información; sin embargo, está fuera del enfoque introductorio de este libro. Asimismo, para mostrar las ventajas y desventajas de ambos paquetes, se presentan los códigos implementados para generar los mismos mapas. Es decisión personal de cada usuario escoger el paquete que más le guste, o usar ambos, dependiendo de lo que quiera hacer.

Para la generación de un mapa (en *Cartopy* y *PyGMT* o cualquier otro paquete), se debe hacer una proyección para *aplanar* el globo terráqueo sobre una superficie (la pantalla del computador o una hoja de papel), lo cual requiere una transformación sistemática de latitud y longitud en la esfera a *(x,y)* en un plano. Es inevitable además que toda proyección de la esfera sobre el plano distorsione la superficie. Dependiendo de lo que se quiera, algunas distorsiones son aceptables y otras no, de modo que es importante saber qué tipo de proyección se quiere utilizar [35, 16].

1. *Cartopy*

1.1. Mapa global

El primer ejemplo es un mapa global generado con el código *cartomap01.py*. Observemos como los comandos son similares a las del capítulo anterior.

cartomap01.py

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
# Proyección PlateCaree
proj0 = ccrs.PlateCarree(central_longitude=180.0)

fig = plt.figure(figsize=(10,8))
ax  = fig.add_subplot(projection=proj0)
ax.coastlines()
ax.set_xticks([-180,-120,-60,0,60,120,180], crs=proj0)
ax.set_yticks([-90,-60,-30,0,30,60,90],crs=proj0)
```

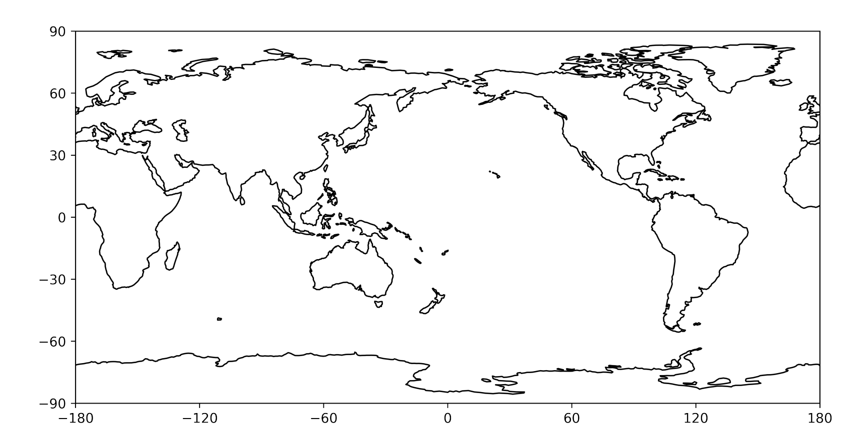


Figura 8.1. Mapa global con *Cartopy* con el código *cartomap01.py*.

Lo primero que cabe señalar es la proyección geográfica que usamos:

```
proj0 = ccrs.PlateCarree(central_longitude=180.0)
...
ax1  = fig.add_subplot(111,projection=proj0)
```

En este caso, el *axes=ax* presenta la proyección de la esfera sobre un plano. En este momento, la figura está vacía. Además, se centra en la longitud deseada *central_longitude=180.0*, o *0.0* si no está definida por el usuario. A continuación, se grafican las líneas de costa:

```
ax.coastlines()
```

Cartopy, de manera automática, crea un borde de la figura, pero no pone los valores en los ejes (latitud y longitud). Esto se puede adicionar con el siguiente procedimiento:

```
ax.set_xticks([-180,-120,-60,0,60,120,180], crs=proj0)
ax.set_yticks([-90,-60,-30,0,30,60,90], crs=proj0)
```

1.2. Proyecciones esféricas y topografía

Hay otro tipo de proyecciones, incluyendo las del globo terráqueo; por ejemplo, proyecciones esféricas como *Mollweide* u *Orthographic*.

cartomap02.py

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt

proj = ccrs.Orthographic(central_longitude=-90.0)
fig = plt.figure(figsize=(10,8))
ax  = fig.add_subplot(111,projection=proj)
ax.stock_img()
ax.gridlines(linewidth=1.0)
```

En estos casos, el mapa puede estar centrado en alguna longitud deseada con *central_longitude*, y funciona de forma similar para la latitud.



Figura 8.2. Mapa con proyección esférica, con el código *cartomap02.py*.

Si se quiere otra proyección, por ejemplo, puede utilizar *ccrs.Mollweide()*:

```
proj = ccrs.Mollweide(central_longitude=-90.0)
```

El resultado es una proyección elíptica muy diferente, en la cual se observan Australia, Europa y África. Más detalles de otras proyecciones se pueden encontrar acá.

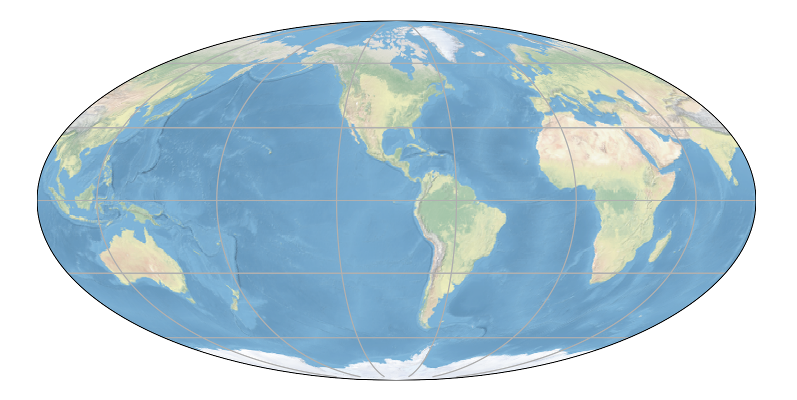


Figura 8.3. Mapa con proyección elíptica, con el código *cartomap02.py*, usando *ccrs.Mollweide()*.

Para la imagen de fondo, que muestra en colores la topografía y batimetría, *Cartopy* tiene con antelación cargada la imagen, y solo la incluye con la proyección deseada, por medio del comando *ax.stock_img()*. Esta es una imagen de baja resolución, así que al hacer *zoom* en alguna parte de la figura, la calidad de la imagen es insuficiente. Más adelante, se expondrá cómo desplegar información topográfica de mayor resolución.

Además, para incluir una grilla visible (por ejemplo, para los meridianos), se solicita con un grosor específico con el siguiente comando:

```
ax.gridlines(linewidth=1.0)
```

1.3. Mapas con fronteras

Los paquetes de mapas, además, tienen la posibilidad de incluir otro tipo de información geográfica, como ríos, lagos, fronteras políticas, etc. El siguiente código, *cartomap03.py*, genera un mapa de Suramérica, que incluye los límites políticos.

cartomap03.py

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import cartopy.feature as cfeature

reg  = [-85, -30, -60, 15]
proj = ccrs.PlateCarree()

fig = plt.figure(figsize=(10,8))
ax  = fig.add_subplot(111,projection=proj)
ax.set_extent(reg)
ax.coastlines()
ax.add_feature(cfeature.BORDERS,linestyle=':')
...
```

Notemos que el código tiene la misma estructura que los anteriores, excepto por la región que se define con el siguiente comando:

```
reg = [-85, -35, -60, 15]
ax.set_extent(reg)
```

En tanto, para adicionar los límites políticos, se carga el módulo de *cartopy.feature* y se agregan las fronteras con una linea punteada de este modo:



Figura 8.4. Mapa de Suramérica con fronteras políticas usando *Cartopy* con el código *cartomap03.py*.

```
import cartopy.feature as cfeature
...
ax.add_feature(cfeature.BORDERS, linestyle=':')
```

1.4. Mapas con colores

En algunos casos, solo se requiere mostrar un mapa con colores, en el cual se separen los cuerpos de agua de los cuerpos de tierra. El código *cartomap04.py* produce dos subplots, uno general en la región *[-170, -100, 20, 60]*, y el otro un zoom en la región *[-130, -120, 46, 52]*. El objetivo es que el mar esté en color azul, y la parte terrestre en otro color. Además, en este programa también se cambia la posición y el tamaño del subplot *ax2*.

cartomap04.py

```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import cartopy.feature as cfeature

reg1 = [-170, -100, 20, 60]
reg2 = [-130, -120, 46, 52]
proj = ccrs.Mercator()

fig = plt.figure(figsize=(12,10))
ax1 = fig.add_subplot(2, 1, 1, projection=proj)
ax1.set_extent(reg1)
ax1.add_feature(cfeature.LAND)
ax1.add_feature(cfeature.OCEAN,color='lightblue')
ax1.coastlines(resolution='50m')
ax1.plot([-130, -130, -120, -120, -130],
         [46, 52, 52, 46, 46],
         color='red', linestyle='--',
         transform=ccrs.PlateCarree(),)
```

```
ax2 = fig.add_subplot(2, 1, 2, projection=proj)
... # posición subplot
ax2.set_extent(reg2)
ax2.add_feature(
            cfeature.GSHHSFeature(scale='auto'),
            facecolor=cfeature.COLORS['land'],)
ax2.patch.set_facecolor('lightblue')
```

Comenzando con la definición de la proyección *ccrs.Mercator()*, se generan los dos subplots de igual manera a como se hace con *Matplotlib*.

```
fig = plt.figure(figsize=(12,10))
ax1 = fig.add_subplot(2,1,1, projection=proj)
...
ax2 = fig.add_subplot(2,1,2, projection=proj)
```

Entonces, para la primera región (la grande), se incluyen *features* de región terrestre y región oceánica, las líneas de costa; y se puede adicionar un cuadro que muestre la zona o región más pequeña:

```
ax1.add_feature(cfeature.LAND)
ax1.add_feature(cfeature.OCEAN,color='lightblue')
ax1.coastlines(resolution='50m')
...
ax1.plot([-130, -130, -120, -120, -130],
         [46, 52, 52, 46, 46],
         color='red', linestyle='--',
         transform=ccrs.PlateCarree(),)
```

Para la segunda región (la pequeña), en principio podríamos hacer lo mismo. Sin embargo, se muestra otra forma de utilizar *features*, que descargan *shapefiles* de mayor resolución de las bases de datos de Natural Earth Data o de la base de datos GSHHS.

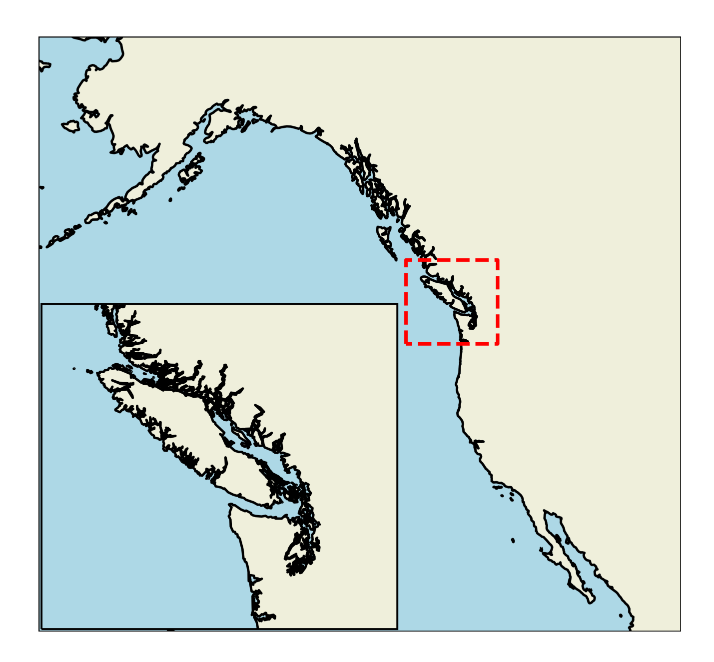


Figura 8.5. Dos mapas que separan la parte oceánica y la parte terrestre con *Cartopy*, mediante el código *cartomap04.py*.

```
ax2 = fig.add_subplot(2, 1, 2, projection=proj)
ax2.set_extent(reg2)
ax2.add_feature(cfeature.GSHHSFeature(scale='auto'),
            facecolor=cfeature.COLORS['land'],)
ax2.patch.set_facecolor('lightblue')
```

Es importante tener en cuenta que, en este caso, se descarga la información de la región terrestre, y se pone el fondo en color azul (el océano). Además, para cambiar la posición y el tamaño del subplot (*ax2*), se puede usar el siguiente ejemplo (un poco complejo):

```
box = ax2.get_position()
box.x0 = box.x0 - 0.01
box.x1 = box.x1 - 0.125
```

```
box.y0 = box.y0 + 0.398
box.y1 = box.y1 + 0.398 - 0.125
ax2.set_position(box)
```

1.5. Ríos y fronteras

La figura 8.6 muestra dos paneles con fronteras políticas (países y estados o departamentos), y ríos principales a escala regional.

cartomap05.py

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# Descargue fronteras y ríos de Natural Earth Data
countries = cfeature.NaturalEarthFeature(
    category='cultural',
    name = 'admin_0_boundary_lines_land',
    scale='10m',
    facecolor='none',)
states = cfeature.NaturalEarthFeature(
    category='cultural',
    name = 'admin_1_states_provinces',
    scale='10m',
    facecolor='none',)
rivers = cfeature.NaturalEarthFeature(
    'physical',
    'rivers_lake_centerlines',
    '10m',)

proj4 = ccrs.Mercator()
fig = plt.figure(figsize=(12,10))
```

```
ax = fig.add_subplot(1, 2, 1, projection=proj4)
ax.set_extent([-85, -60, -6, 13])

# Adicionar países y departamentos
ax.add_feature(cfeature.OCEAN,color='lightblue')
ax.add_feature(states,edgecolor='lightgrey',
        linestyles='-')
ax.add_feature(countries,edgecolor='k')
ax.coastlines(resolution='10m')

ax2 = ...

# Adicionar rios
...
ax2.add_feature(rivers,edgecolor='blue',
        facecolor='none')
...
```

El código *cartomap05.py* inicia por descargar la información de *shapefiles* con la información de fronteras y ríos:

```
countries = cfeature.NaturalEarthFeature(
    category='cultural',
    name = 'admin_0_boundary_lines_land',
    scale='10m',
    facecolor='none', )
```

```
states = cfeature.NaturalEarthFeature(
    category='cultural',
    name = 'admin_1_states_provinces',
    scale='10m',
    facecolor='none',)
```

```
rivers = cfeature.NaturalEarthFeature(
     'physical',
     'rivers_lake_centerlines',
     '10m',)
```

Con ello, se define la categoría (*cultural* o *physical*), la escala (*10m*) y el nombre del *shapefile* (*admin_0...*).

De forma similar, se ejecuta el procedimiento para los ríos, una propiedad física, con la resolución deseada. En este caso, *10m* equivale a una escala *1:10.000.000*. A su vez, las líneas de costa también se pueden solicitar con la resolución deseada, con el comando *ax.coastlines(resolution='10m')*.

La información de categorías físicas y culturales se puede encontrar en la página de Natural Earth. Recordemos que es importante conocer el nombre exacto de los parámetros para que Python pueda encontrar la información.

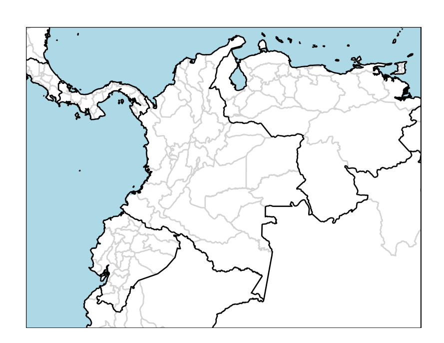
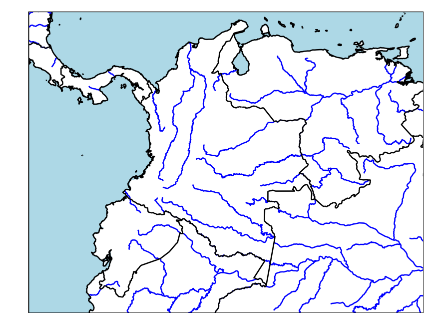



Figura 8.6. Mapa de Colombia con límites (izquierda) y ríos (derecha) en el código *cartomap05.py*.

Cartopy permite además acceder a bases de datos como GSHHS, facilitan-do la descarga de información de costas, fronteras, ríos, etc., con comandos como:

```
coast = cfeature.GSHHSFeature(scale='i',
     levels= [1],
     edgecolor='k',)
```

1.6. ¿Cómo incluir datos propios?

Como último ejemplo, muchas veces es importante incluir la **topografía** en el mapa, ya que puede mostrar algunos patrones importantes que se quiera resaltar. Además, con frecuencia se desea desplegar **información adicional** (puntos donde se tomaron datos, una ubicación, etc.), es decir **datos**.

Así, los datos en el archivo *rsnc.dat* corresponden a la latitud y la longitud de las estaciones de la Red Sismológica Nacional de Colombia (RSNC), administrada por el Servicio Geológico Colombiano (sgc.gov/sismos). El programa *cartomap06.py* genera un mapa, con topografía en el fondo, de la RSNC en el territorio colombiano.

cartomap06.py

```
"""
cartomap06.py
 Con topo, y estaciones lat/lon
...
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature

# lectura del archivo
fname = 'data/rsnc.dat'
...

wms_server = "https://ows.terrestris.de/osm/service?"

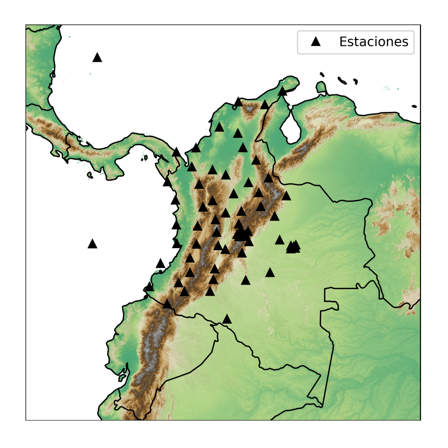
proj = ccrs.Mercator()
fig = plt.figure(figsize=(12,10),dpi=800)
ax  = fig.add_subplot(121,projection=proj)
ax.set_extent([-85, -65, -5, 15])

# Adicionar topografía
```

```
ax.add_wms(wms_server,
           layers=["SRTM30-Colored-Hillshade"])
ax.plot(lon,lat,'k^',label='Estaciones',
   transform=ccrs.Geodetic())
ax.legend()

ax2  = fig.add_subplot(122,projection=proj)
ax2.set_extent([-76, -70, 9, 12])
ax2.add_wms(wms_server,
            ...)

ax2.plot(lon,lat,'k^',transform=ccrs.Geodetic())
```



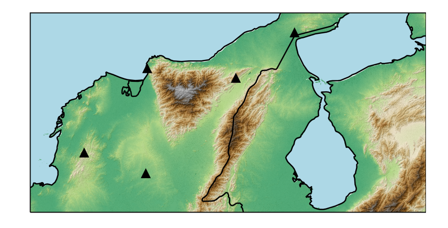


Figura 8.7. Mapa de *Cartopy* de las estaciones de la RSNC con fronteras y topografía. El mapa es el resultado de correr el código *cartomap06.py*.

El paquete *Cartopy* tiene en *ax.stock_img()* la imagen de baja resolución de la topografía (y batimetría); pero, si estamos mirando escalas regionales, esta resolución es insuficiente. Para obtener datos de la topografía de mayor resolución, se debe buscar una fuente de imágenes topográficas. Cartopy permite descargar archivos tipo *tile* de la siguiente manera:

```
import cartopy.io.img_tiles as cio
...
tiler = cio.Stamen('terrain-background')
tiler = cio.GoogleTiles(style='satellite')
```

Sin embargo, el servidor *stamen* ya no está disponible y *GoogleTiles* permite desplegar imágenes satelitales, pero las imágenes de topografía no son tan buenas.

Una alternativa es utilizar servidores WMS (*Web Map Service*), que tienen imágenes que contienen la información de topografía y batimetría de alta resolución. La primero que se necesita es la dirección de un servidor WMS, por ejemplo:

```
wms_server = "https://ows.terrestris.de/osm/service?"
```

Posteriormente podemos incluir la imagen con la proyección adecuada, así:

```
proj = ccrs.Mercator()
fig = plt.figure(figsize=(12,10),dpi=800)
ax  = fig.add_subplot(121,projection=proj)
ax.set_extent([-85, -65, -5, 15])

# Adicionar topografía
ax.add_wms(wms_server,
           layers=["SRTM30-Colored-Hillshade"],
          )
```

Acá, el parámetro *layers* contiene el nombre de la capa que se quiere incluir en el mapa, en nuestro caso topografía con sombreado en colores.

Por último, se pueden incluir las posiciones de las estaciones mediante el comando *ax.plot()*, tal como se haría en *Matplotlib*; sin embargo, se debe incluir una transformación de los valores de *lat* y *lon*, para proyectarlos en el mapa:

```
ax.plot(lon,lat,'k^',
      transform=ccrs.Geodetic(),
      label='Estaciones'
      )
...
ax2.plot(lon,lat,'k^',transform=ccrs.Geodetic())
```

Observemos como se incluye la leyenda explicativa, y se pueden utilizar los diferentes parámetros disponibles en *plt.plot()*. Es posible utilizar además otros comandos de gráficas como *ax.scatter()*, *ax.quiver*, *ax.contourf()*, etc.

2. *PyGMT*

El paquete de *PyGMT* no está basado en *Matplotlib* y, dado que se sustenta en *GMT*, las figuras que se producen tienen el formato *PS* o *postscript*, dado que este no tiene la capacidad de interacción, que sí tiene *Cartopy*. Sin embargo, *PyGMT* permite guardarlas en formato *PDF*.

2.1. Mapa global

El primer ejemplo da lugar a un mapa a escala global con *GMT*, por medio del código *gmtmap01.py*.

gmtmap01.py

```
import pygmt
# Proyección Cilíndrica
proj = 'Cyl_stere/180/0/8i'

fig = pygmt.Figure()
fig.coast(
    region='g',
    projection=proj,
```

```
    shorelines=True,
    water=False,
    land=False,
    frame=True,)
fig.savefig('../figs/chap08_map07.pdf')
fig.show()
```

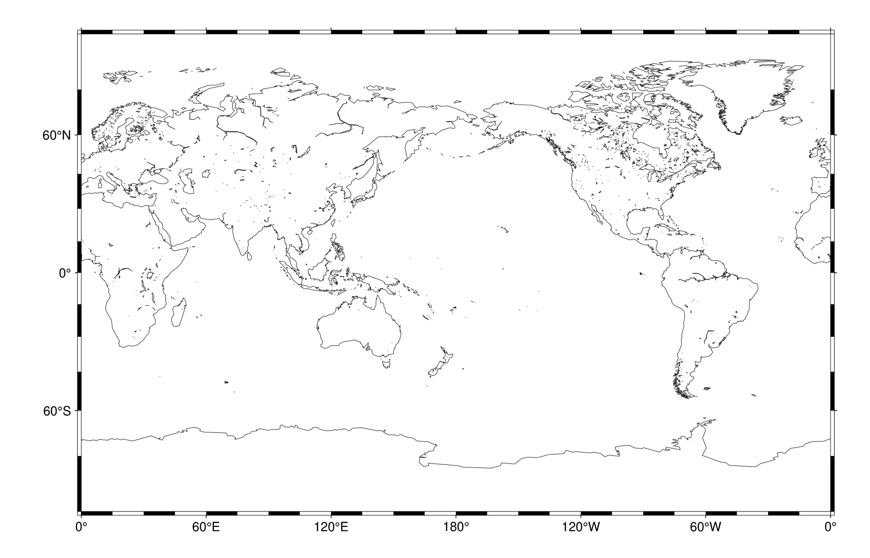


Figura 8.8. Mapa global con *PyGMT* con el código *gmtmap01.py*.

Para comenzar, se debe importar el paquete con *import pygmt*. La proyección utilizada en este caso es cilíndrica estereográfica:

```
proj = 'Cyl_stere/180/0/8i'
```

Dicha proyección está centrada en las coordenadas de longitud 180 y latitud 0. En tanto, el tamaño de la figura es de *8i*, ocho pulgadas. Para iniciar la figura, se utiliza el siguiente procedimiento:

```
fig = pygmt.Figure()
```

Observemos la diferencia con *Matplotlib*. También podemos ver que el comando *coast* dibuja las líneas de costa, utilizando la región *global* o *'g'*, así:

```
fig.coast(
    region='g',
    projection=proj,
    shorelines=True,
    water=False,
    land=False,
    frame=True,
)
```

El comando también permite definir regiones por siglas de países, estados (de EE.UU.), o definir los límites *[lon1, lon2, lat1, lat2]*. Se debe definir también la proyección, y, si se quiere, es posible indicarle a Python que trace las líneas de costa con *shoreline=True*. Se le puede solicitar que se coloree el agua o la región terrestre, aunque en este ejemplo, aquello no se hizo. Por ejemplo, para colorear el mar de azul se puede usar *water='lightblue*.

Por último, es **necesario** solicitar explícitamente que la figura sea mostrada en Jupyter Notebooks con el siguiente comando:

```
fig.show()
```

Finalmente, se le puede solicitar que guarde el archivo de la figura en formato *PDF* con:

```
fig.savefig('nombre.pdf')
```

2.2. Proyecciones esféricas y topografía

Al igual que con *Cartopy*, el programa *gmtmap02.py* genera un mapa global, con una proyección esférica (ortográfica, *'-G'* o Mollweide, *'Moll'*), centrada en Suramérica, con topografía incluida:

gmtmap02.py

```
import pygmt

proj = 'G-70/0/4.5i'
fig = pygmt.Figure()
fig.grdimage(
    '@earth_relief_30m',
    region='g',
    projection=proj,
    cmap='globe',
    shading=True,)
fig.show()
```

En el código anterior, se solicitó la proyección ortográfica (figura 8.9) con *G-70/0/4.5i*, centrada en longitud *-70*, y latitud *0*, y genera una figura de 4.5 pulgadas. Si se quiere otra proyección, por ejemplo, se puede utilizar la proyección elíptica *Mollweide* que se muestra en la figura 8.10. Usamos:

```
proj = 'Moll/-70/4.5i'
```

Para la imagen de fondo, se usa el comando *grdimage*:

```
fig.grdimage(
    '@earth_relief_30m',
    region='g', projection=proj,
    cmap='globe', shading=True,)
```

Para la imagen de topografía, el archivo a utilizar es *@earth_relief_30m*, el cual es un archivo en la base de datos de *GMT* que es descargado. Es decir, el computador va a bajar el archivo de internet, y lo colocará en una ubicación conocida por *PyGMT*. El archivo descargado es entonces un archivo en formato de arreglo o grilla, y le podemos pedir que use una escala de colores (*colormap*) predefinida por GMT (*cmap='globe'*), e incluso ponerle sombra (*shading*).



Figura 8.9. Mapa con proyección esférica con *PyGMT* por medio del código *gmtmap02.py*.

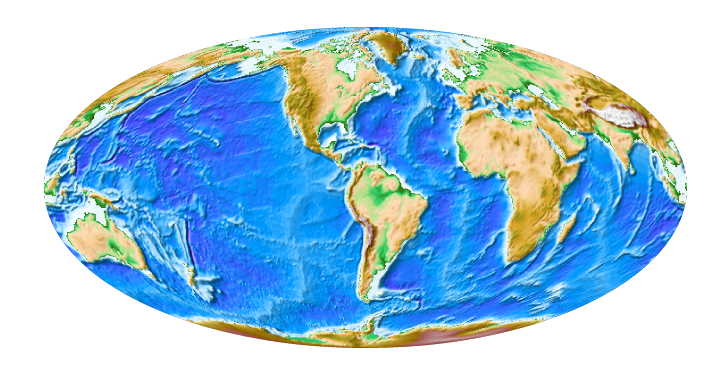


Figura 8.10. Mapa con proyección elíptica con *PyGMT* por medio del código *gmtmap02.py*, usando *proj = 'Moll/-70/4.5i'*.

Es importante señalar que los datos con la topografía y batimetría global tienen multiples resoluciones. En ese sentido, se debe tener cuidado de no usar una resolución muy alta para la escala global, ya que el archivo sería muy pesado (cientos de GB). Las resoluciones disponibles son las siguientes:

- Para bajas resoluciones (*non-tiled*) *['01d', '30m', '20m', '15m', '10m', '06m']*
- Para altas resoluciones (*tiled*) *['05m', '04m', '03m', '02m', '01m', '30s', '15s', '03s', '01s']*

Las bases de datos de alta resolución están *tiled* o subdivididos en partes (por parches) como en *Cartopy*. En otras palabras, si se solicita un archivo de alta resolución, *PyGMT* solo baja las partes que necesita, con base en la región que se va a graficar.

Los *colormaps* automáticos de *GMT* se pueden encontrar acá, pero es posible diseñar un *colormap* si así se decide (aunque no lo vamos a ver en este texto). Algunos *colormap* usados en geociencias incluyen *seis*, *polar*, *gray*, *etopo1*, y muchos más.

2.3. Mapas con fronteras

PyGMT ofrece la posibilidad de incluir otro tipo de información geográfica como ríos, lagos, fronteras políticas, etc. El programa *gmtmap03.py* genera un mapa de Suramérica (figura 8.11) que incluye los límites políticos.

gmtmap03.py

```
import pygmt

reg  = [-85, -35, -60, 15]
proj = 'M0/0/4i'

fig = pygmt.Figure()
fig.coast(
    region=reg,
    projection=proj,
```

```
    shorelines=True,
    frame=True,
    resolution='l',
    borders=1,)
fig.show()
```

Este mapa usa las mismas herramientas anteriores; sin embargo, mediante el comando *coast* se pueden solicitar, además de las costas, los límites políticos con *borders=1*. En este caso, el número *1* representa las fronteras políticas de los países. Para departamentos o estados, el código es *2*, y, en algunos casos, el *3* puede mostrar fronteras entre condados. Para incluir varios límites políticos (paises, estados/provincias/departamentos, etc.), usamos *borders=[1,2,3]*, y si se desea mayor control del formato (color y tipo de lineas), se puede utilizar el siguiente comando:

```
["1/0.5p,white", "2/0.5p,red", "3/0.5p,blue"]
```

Además, se define la resolución del *shapefile* que tiene las líneas de costa con *resolution='l'*, que, en este caso, por ser un mapa general, usamos con baja resolución (*l=low*). Hay cinco niveles de resolución (con un cambio de 80 % entre niveles):

- *c: crude*
- *l: low (default)*
- *i: intermediate*
- *h: high*
- *f: full*

Se recomienda no usar en seguida la resolución máxima, ya que los *shapefile* pueden ser muy pesados.

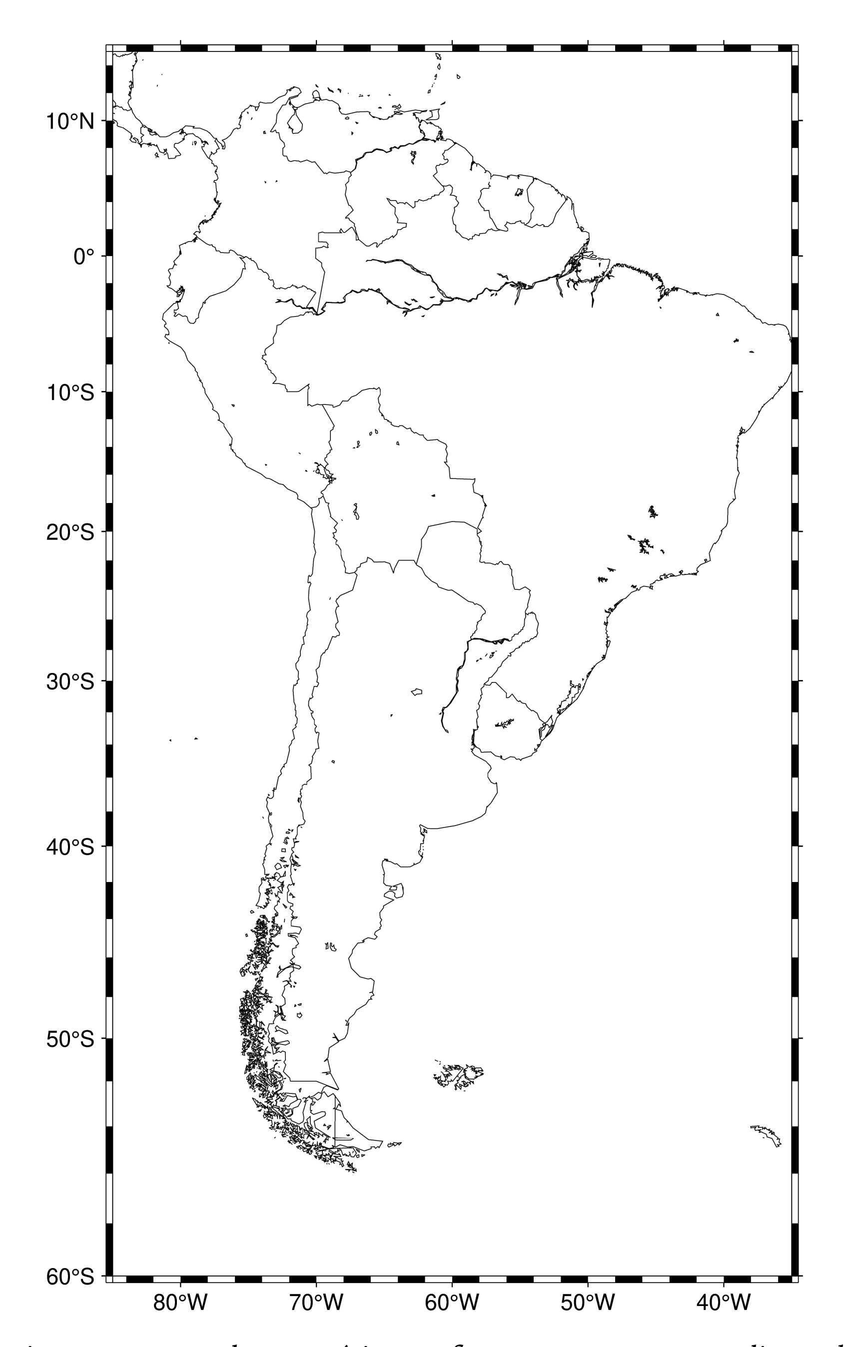


Figura 8.11. Mapa de Suramérica con fronteras con *PyGMT* mediante el código *gmtmap03.py*.

2.4. Mapas con colores

A veces, solo queremos mostrar un mapa con colores, en el cual se separen los cuerpos de agua de la tierra. El código *gmtmap04.py* genera dos mapas (figura 8.12), uno a escala regional, y el otro a escala local (un zoom), en una región costera. El objetivo es que el mar se muestre en color azul, y la región terrestre, en otro color.

gmtmap04.py

```
import pygmt

reg1 = [-170, -100, 20, 60]
reg2 = [-130, -120, 46, 52]

fig = pygmt.Figure()
fig.coast(
    region=reg1,
    projection='M8i',
    shorelines=True,
    water='lightblue',
    land='grey',
    frame=True
)
fig.plot([-130, -120, -120, -130, -130],
         [  46,   46,   52,   52,   46],
         pen="2p,red,-"
)

pygmt.config(MAP_FRAME_TYPE="plain")
fig.shift_origin(xshift='0.2i',yshift='0.1i')
fig.coast(
    region=reg2,
    projection='M4i',
```

```
    shorelines=True,
    water='lightblue',
    land='grey',
    frame='f'
)
fig.show()
```

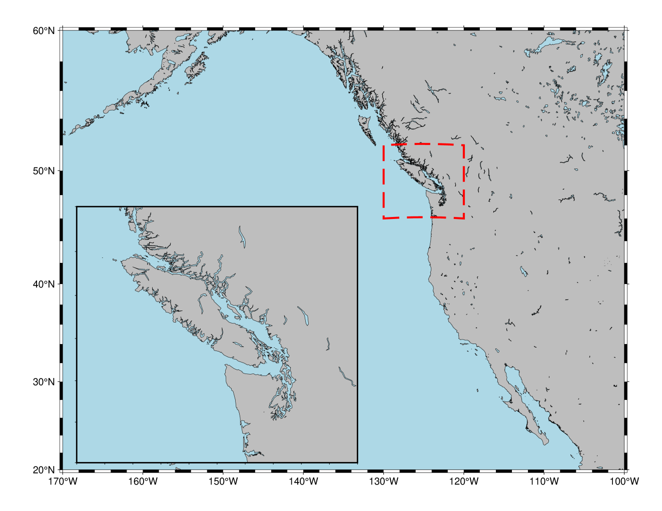


Figura 8.12. Dos mapas que separan la región oceánica y la región terrestre con *PyGMT*, creados por medio del código *gmtmap04.py*.

La proyección en ambas figuras será Mercator ('M8i', 'M4i'), de tal forma que los mapas tendrán tamaños distintos. Como *PyGMT* no usa el mismo modelo de *Matplotlib*, las dos figuras se crean de la siguiente manera:

```
fig = pygmt.Figure()
fig.coast(
...
)
fig.shift_origin(xshift='0.2i',yshift='0.1i')
```

Así, usando la misma figura, *fig*, se redefine el origen. *GMT* genera las figuras en el orden en que se programa; es decir, si se hace primero la figura pequeña y luego la grande, la grande estará encima y no se verán las dos.

Ambas figuras tienen el mismo patrón, y la región acuática y la terrestre se colorean con el siguiente comando:

```
fig.coast(...,
    water='lightblue',
    land='grey',
    frame='f')
```

Tenga en cuenta que el parámetro *frame* define si hay un cajón que encierre la figura, y si tiene o no valores en los ejes *x* o *y*. Para generar un *frame* más sencillo en el *subplot* pequeño, como el mostrado en la figura 8.12, se debe cambiar la configuración *default* de *PyGMT* con el siguiente comando:

```
pygmt.config(MAP_FRAME_TYPE="plain")
```

Estos cambios solo aplican durante la ejecución del código. Si se desea un cambio permanente, este se puede obtener al modificar los parámetros básicos de *PyGMT* en el computador del usuario.

2.5. Ríos y fronteras

El programa *gmtmap05.py* genera un mapa como el que se muestra en la figura 8.13 para Colombia y sus alrededores, incluyendo límites políticos de países y departamentos (estados/provincias), y los ríos principales en color azul.

gmtmap05.py

```
import pygmt
reg  = [-85, -60, -6, 13]
proj = 'M4i'
fig = pygmt.Figure()
fig.coast(
```

```
    region=reg,
    projection=proj,
    shorelines=True,
    frame='f', resolution='l',
    borders=['1/1p,black','2/1p,gray'],
    water='lightblue',
    land="white",)

fig.shift_origin(xshift='4.2i')  # correr prox figura
fig.coast(
    region=reg,
    projection=proj,
    shorelines=True,
    frame='f', resolution='i',
    borders=['1/1p,black'],
    rivers= ['1/blue','2/blue','3/blue'],
    water='lightblue',
    land="white",)
fig.show()
```

Para extraer la información del *shapefile* de fronteras y ríos, dentro del comando *coast*, se utiliza el siguiente procedimiento:

```
fig.coast(...
    borders=['1/1p,black','2/1p,gray'],
    water='lightblue',
    land="white",)
fig.coast(...
    rivers= ['1/blue','2/blue','3/blue'],
...)
```

El comando en ambos casos tiene un formato específico de *GMT* con el patrón *1/1p,color*, de modo que el primer número es el tipo de característica

(ríos principales, límites de países, o ríos secundarios y límites departamentales). La segunda parte es la descripción de las líneas, su tamaño y color. En tanto, si se requiere mayor precisión en límites políticos o ríos, puede solicitarla mediante el siguiente comando:

```
borders=['1/1p,black','2/1p,gray']
```

O bien:

```
rivers= ['1/blue','2/blue','3/blue','4/blue']
```

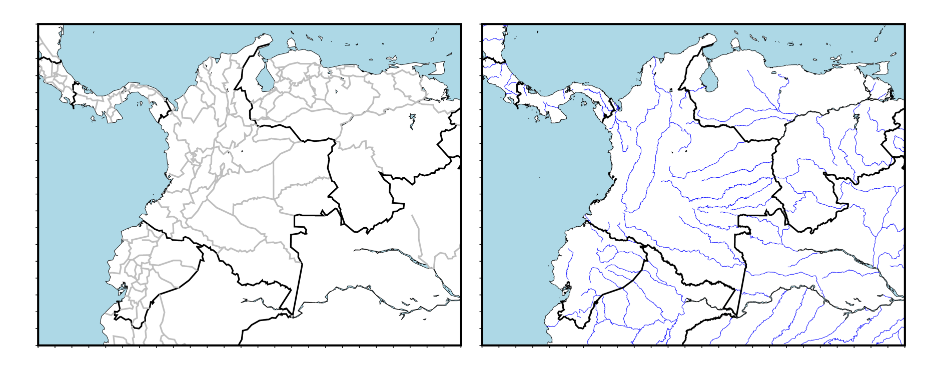


Figura 8.13. Mapa de Colombia con límites (izquierda) y ríos (derecha) en el código *gmtmap05.py*.

2.6. ¿Cómo incluir datos propios?

Como último ejemplo, muchas veces es importante incluir la **topografía** en el mapa, ya que puede mostrar algunos patrones importantes que se quieran destacar. Además, con frecuencia se desea desplegar **información adicional** (puntos donde se tomaron datos, una ubicación, etc.); es decir, **datos**.

El programa *gmtmap06.py* carga los datos del archivo *rsnc.dat* y los organiza en arreglos con latitud y longitud de las estaciones sismológicas de la RSNC. A continuación, hace un mapa (figura 8.14), con topografía en el fondo, de la RSNC en el territorio colombiano a escala regional y local.

gmtmap06.py

```
import pygmt
fname = 'data/rsnc.dat'
...
reg   = [-85, -65, -5, 15]
reg2  = [-76, -70,  9, 12]
proj  = 'M6i'

fig = pygmt.Figure()
pygmt.config(MAP_FRAME_TYPE="plain")
fig.grdimage(
    '@earth_relief_05m',
    region=reg,
    projection=proj,
    cmap='etopo1',
    shading=True,)
fig.coast(
    region=reg,
...)

fig.plot(lon,lat,
    style='t0.15i',
    color='black',
    label='Estaciones',)
fig.legend()

fig.shift_origin(xshift='4.5i',yshift='1.5i')
fig.grdimage(
    '@earth_relief_15s',
    region=reg2,
    projection=proj,
    cmap='etopo1',
```

```
    shading=True,)
fig.coast(
    region=reg2,
...
fig.plot(lon,lat,
...
fig.show()
```

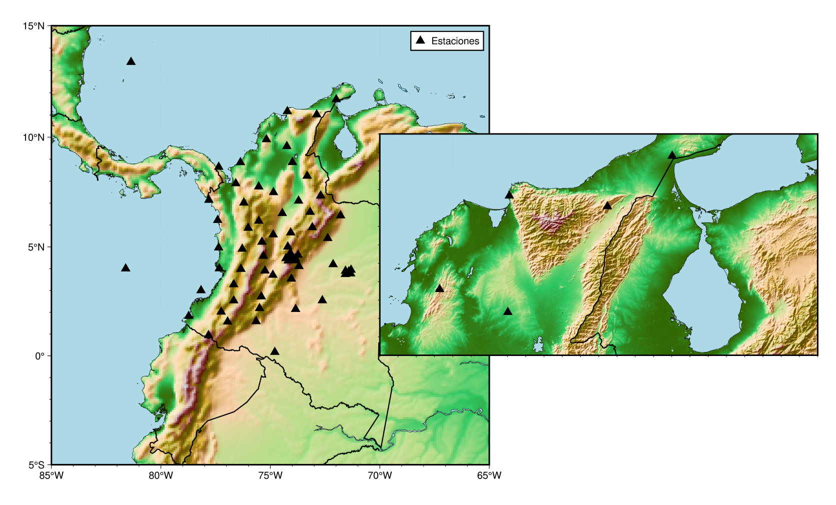


Figura 8.14. Mapa de *PyGMT* de estaciones sismológicas de la RSNC con fronteras y topografía. El mapa es el resultado de correr el código *gmtmap06.py*.

El programa *gmtmap06.py* combina los diferentes comandos ya utilizados. Primero, para la topografía se utiliza *grdimage*:

```
fig.grdimage(
    '@earth_relief_05m',
    ...
    cmap='etopo1',
    shading=True,)
```

Acá, se incluye la topografía con resolución suficiente para la escala regional, incluyendo sombras. Dado que para la segunda región se requiere mayor resolución, entonces:

```
fig.grdimage(
    '@earth_relief_15s',
    ...)
```

Tal como en los mapas anteriores, se incluyen límites políticos y líneas de costa con el comando *coast*. Observemos como el agua se representa en color azul. Si no se pone *water='lightblue'*, la **batimetría se desplegará** en la figura.

Por último, se despliegan las estaciones *(lon,lat)*, con lo que se indica que estas se ponen de color negro y en forma de triángulos.

```
fig.plot(lon,lat,
    style='t0.2i',
    color='black',
    label='Estaciones',)
```

Finalmente, se despliega la leyenda con el comando *fig.legend()*.

3. ¿Cuál escoger?

Antes de contestar, es importante recordar que ambos paquetes están en etapa de desarrollo. Por consiguiente, en el futuro, pueden mejorar mucho o hacerse muy complicados. En la actividad de investigación y presentación de resultados puede haber muchas cosas que se quiere hacer cuando se desea desplegar información en mapas. Asimismo, en los ejemplos mostrados en este libro, se usó en ambos casos información de *shapefiles* y *tiles* disponibles; por consiguiente, no se discute cómo incluir datos que el usuario pueda tener. Con ello se alude a información de *shapefiles* de carreteras o caminos, o grillas (por ejemplo, en formato *NetCDF*), con datos que el usuario haya tomado en campo (gravimetría, sísmica, etc.).

En ambos paquetes es posible desplegar esta información; sin embargo, esto está fuera del enfoque introductorio de este libro. El lector puede averiguar cómo incluir sus datos y ponerlos en un mapa con *Cartopy* o con *PyGMT*. Ambos tienen ventajas y desventajas, y quizá, por ahora, es cuestión personal por cuál decidirse.

Problemas

Para cada problema elija el paquete de mapas que prefiera.

1. Una base de datos [18] con información de precipitación, temperatura y radiación, en más de 2600 cuencas, está disponible en la página del U.S. Geological Survey (sciencedatabase.gov). El archivo con la ubicación de cada estación, *BASIN_CHARACTERISTICS.csv*, incluye latitud, longitud, elevación y área de cada cuenca.

    a) Genere un mapa con la ubicación de las estaciones o cuencas a nivel global.

    b) Escoja una estación al azar, márquela en el mapa anterior, y haga un mapa con un zoom (por ejemplo, 15 grados a cada lado). Luego, agregue la topografía. Busque una buena forma de presentar el resultado de manera amigable.

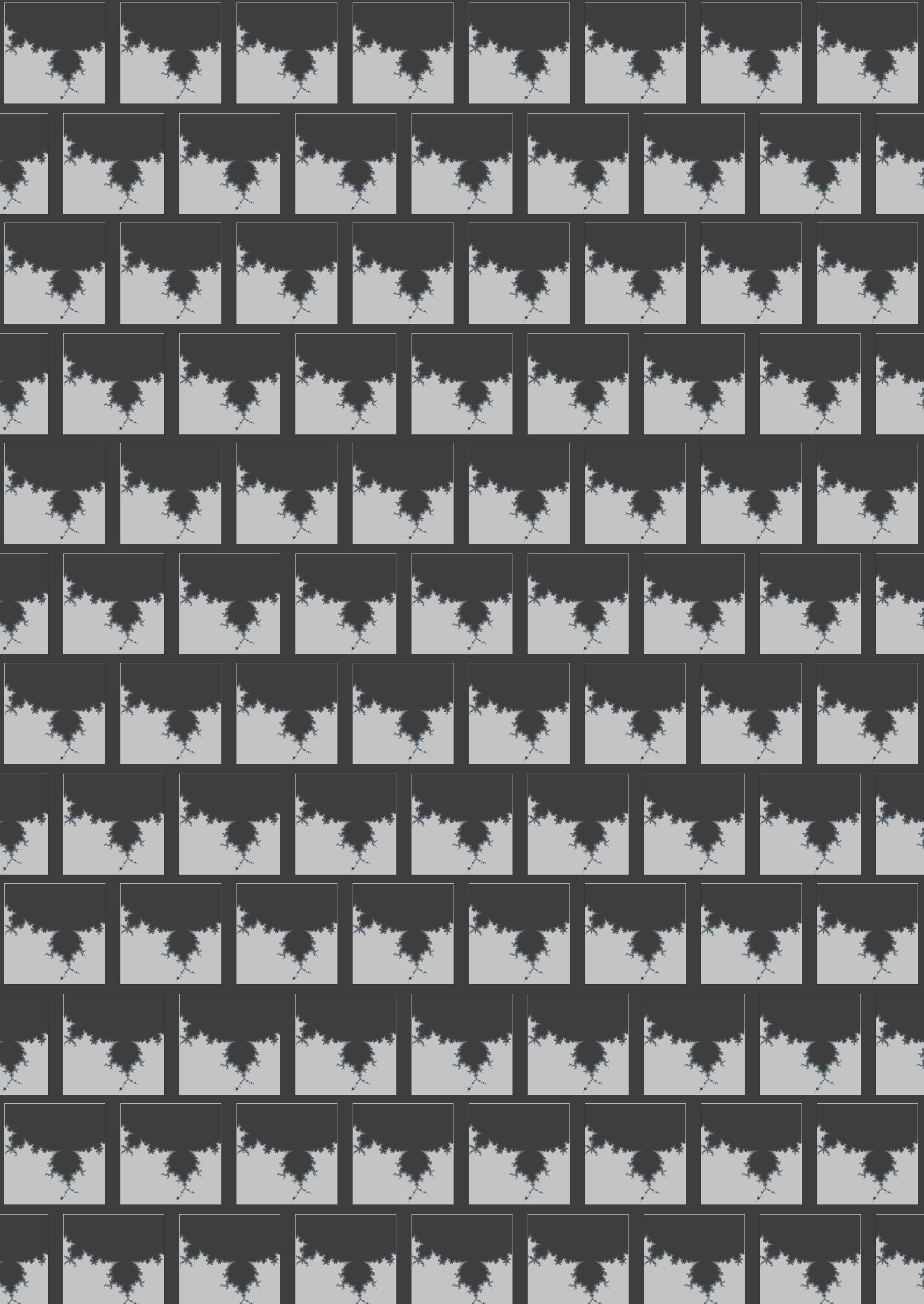
2. En algunos casos, es importante presentar un conjunto de datos espaciales que tienen varios parámetros. Por ejemplo, para los terremotos, se definen parámetros como latitud, longitud y profundidad, pero adicionalmente, magnitud. El archivo *eq_japan.csv* contiene información acerca de los terremotos ubicados en la zona de Japón desde 2018, de acuerdo con el catálogo del USGS (earthquake.usgs.gov/earthquakes/search/, último acceso, septiembre, 2021).

Haga un mapa de la zona, e incluya la ubicación de los terremotos, incorporando la información de magnitud y profundidad. Por ejemplo, se puede definir una escala de colores para profundidad, y usar la magnitud para asignar el tamaño del símbolo.



Capítulo 9

Números complejos



Las geociencias son un campo amplio y multidisciplinario, y, por consiguiente, las herramientas matemáticas requeridas para estudiar, modelar y analizar los procesos en la Tierra también lo son. Aunque el uso de números complejos puede no ser familiar para muchos practicantes de ciencias de la Tierra, estos son fundamentales en el tratamiento de datos espaciales y series de tiempo; en efecto, los números complejos se usan para el análisis de Fourier y varios campos de la geofísica y geología estructural, por mencionar algunos [11, 33].

Aunque no es la intención de este libro hacer un repaso de los números complejos, sí es necesario definir el número complejo z como:

$$z = x + iy, \tag{9.1}$$

donde el número imaginario i es:

$$i = \sqrt{(-1)},$$

o bien:

$$i^2 = -1$$

Para el número complejo z, se tiene una parte real $x = Re(z)$, y una parte imaginaria $y = Im(z)$. Tengamos en cuenta que la variable y es una variable real. El complejo conjugado de z, se define así:

$$z^* = x - iy \tag{9.2}$$

Un número complejo se puede representar de forma gráfica como un punto en un plano complejo (figura 9.1) con un eje *real* (horizontal) y un eje *imaginario* (vertical). En coordenadas cartesianas, esto está definido por x y y. También se pueden utilizar coordenadas polares para definir el número complejo con su fase θ y su magnitud $r = |z|$. Estas dos formas de definir el número complejo están relacionadas:

$$z = re^{i\theta} = r(\cos\theta + i\sin\theta) = x + iy, \tag{9.3}$$

A la magnitud $|z|$ también se le conoce como el *valor absoluto de* z, y se puede obtener del siguiente modo:

$$zz^* = (x + iy)(x - iy) = x^2 + y^2 = |z|^2, \tag{9.4}$$

o con coordenadas polares:

$$zz^* = re^{i\theta}re^{-i\theta} = r^2e^{i\theta-i\theta} = |z|^2. \qquad (9.5)$$

A su vez, la fase se puede obtener con:

$$y/x = \tan\theta \qquad (9.6)$$

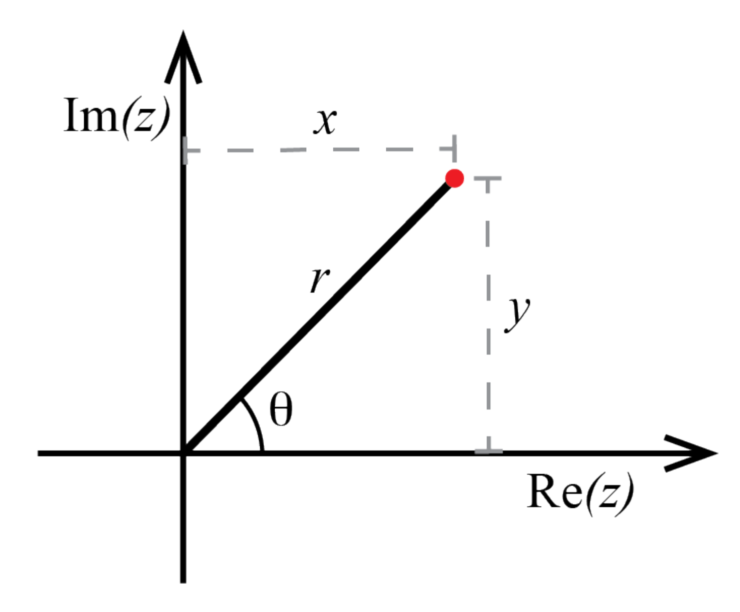


Figura 9.1. Representación del número complejo z en el plano complejo.

1. Números complejos en Python

En algunos lenguajes, el uso de números complejos no es sencillo. En Fortran, por ejemplo, un número complejo es estándar, mientras que en C/C++ no lo es, y se requiere cargar librerías para poder trabajar con ellos. En Python, se puede definir un número complejo usando uno de los siguientes comandos:

```
z = complex(2,3)
z = 2+3j
```

Acá, *j* se usa en vez de *i* en Matlab. Algunos ejemplos básicos del uso de números complejos en Python son los siguientes:

complejo1.py

```
x  = 1
z0 = 2j
z1 = 3  + 4j

print('Real , x)
print(('Imaginario', z0)
print(('Complejo', z1)
print('Parte real e imag de z1 = ',z1.real, z1.imag)
```

```
Real 1
Imaginario 2j
Complejo (3+4j)
Parte real e imag de z1 =  3.0 4.0
```

El anterior código muestra cómo Python despliega los números reales (*x*), imaginarios (*z0*) o complejos (*z1*). Abajo, mostramos dos formas de definir el mismo número complejo:

complejo2.py

```
z0 =  complex(2,3)
z  = 2+3j

print(z, z0)
```

```
(2+3j) (2+3j)
```

1.1. División de números complejos

Supongamos que se requiere dividir dos números complejos. Esto se puede hacer a mano, multiplicando arriba y abajo por el conjugado complejo del denominador, como en el siguiente ejemplo:

$$\begin{aligned} \frac{3i}{5-2i} &= \frac{3i}{5-2i}\frac{(5+2i)}{(5+2i)} \\ &= -\frac{6}{29}+\frac{15}{29}i \\ &= 0.206+0.517i \end{aligned}$$

O bien, se puede hacer con Python:

div_complex.py

```
z0 = 3j
z1 = 5-2j
z2 = z0/z1
print(z2)
```

```
(-0.206...+0.517...j)
```

1.2. Otras operaciones con complejos

Muy similar a lo que puede hacer Python con números reales, el programa *test_complex.py* ejecuta operaciones como multiplicar, u obtener el seno o el valor absoluto de un número complejo. En el siguiente ejemplo, el programa utiliza *NumPy*, aunque también se podría utilizar el paquete *cmath*. La ventaja de usar *NumPy* es que se adapta fácilmente al uso de arreglos de números complejos.

test_complex.py

```
# testcomplex.py
import numpy as np
```

```
zreal = np.random.randint(1,5)
zimag = np.random.randint(1,5)
a = complex(float(zreal),float(zimag))
b = complex(2,1)
c    = a*b
csq  = np.sqrt(c)
csin = np.sin(c)
cexp = np.exp(c)

print('# complejo a    =', a)
print('# complejo b    =', b)
print('c = a x b       =', c)
print('|c|             = %.2f' % abs(c))
print('sqrt(c)         = ',csq)
print('sin(c)          = ',csin)
print('exp(c)          = ',cexp)
```

Cabe anotar que para imprimir de una manera amable el número complejo de salida, se puede solicitar un formato específico, de este modo:

```
print('sqrt(c) = %.2f, %.2fj)' %(csq.real,csq.imag)
```

El resultado, entonces, es el siguiente:

```
# complejo a    = (1+3j)
# complejo b    = (2+1j)
c = a x b       = (-1+7j)
|c|             = 7.07
sqrt(c)         = (1.74, 2.01j)
sin(c)          = (-461.39, 296.26j)
exp(c)          = (0.28, 0.24j)
```

2. Arreglos de números complejos

NumPy tiene la capacidad de trabajar con arreglos de números complejos, lo cual puede ser de gran utilidad. Incluso las operaciones en *NumPy* trabajan sin problema con arreglos complejos.

A continuación, un ejemplo simple de creación de un arreglo complejo y el uso de multiplicación matricial *matmul()* con arreglos complejos.

array_complex.py

```
# array_complex.py
import numpy as np

a = np.array([1+2j, 3+4j, 5+6j])
a = a[:, np.newaxis]
b = a.T

print('Arreglo a, shape ',np.shape(a))
print( a)
print('Arreglo b, shape ',np.shape(b))
print( b)
c = np.matmul(a,b)

print ('matmul(a*a.T)')
print(c)

c = np.matmul(a,np.conjugate(b))

print ('matmul(a*conj(a.T))')
print(c)

c = np.matmul(a,np.conjugate(b))
```

```
print ('sqrt(c)')
np.set_printoptions(precision=3)
print(np.sqrt(c))
```

Lo anterior produce el siguiente resultado:

```
Arreglo a, shape  (3, 1)
[[1.+2.j]
 [3.+4.j]
 [5.+6.j]]
Arreglo b, shape  (1, 3)
[[1.+2.j 3.+4.j 5.+6.j]]
(a*a.T)
[[ -3. +4.j  -5.+10.j  -7.+16.j]
 [ -5.+10.j  -7.+24.j  -9.+38.j]
 [ -7.+16.j  -9.+38.j -11.+60.j]]
(a*conj(a.T))
[[ 5.+0.j 11.+2.j 17.+4.j]
 [11.-2.j 25.+0.j 39.+2.j]
 [17.-4.j 39.-2.j 61.+0.j]]
sqrt(a*conj(a.T))
[[2.236+0.j    3.33 +0.3j   4.151+0.482j]
 [3.33 -0.3j   5.   +0.j    6.247+0.16j ]
 [4.151-0.482j 6.247-0.16j  7.81 +0.j   ]]
```

3. Fractales

En geociencias, los fractales y el concepto de autosimilitud tienen múltiples aplicaciones [38, 32], incluyendo terremotos [29], fallas y fracturas [5], geobiología [13], geomorfología [23], y muchos otros campos [2].

Recomendamos buscar en internet imágenes de fractales. Para generar este tipo de figuras, se requiere del manejo de números complejos. La idea

básica detrás del cálculo de fractales es que hay ciertas funciones complejas que, cuando se calculan de manera repetida, pueden converger (siendo estables), o divergir. El que diverjan o no es muy sensible a pequeños cambios en el valor del número complejo que inicia el cálculo, de ahí su uso en teoría del caos, por ejemplo [39]. Esa transición entre divergencia o estabilidad se observa en ciertas regiones en el plano complejo.

Uno de los sets más famosos es el *conjunto de Mandelbrot*, el cual es relativamente fácil de generar en un programa de computador. Para generar este *conjunto*, empezamos considerando un número complejo c, al cual se le aplica el siguiente algoritmo:

```
comience con z=0

calcule repetidamente z = z*z + c
verifique si |z| > 2 y cuantas repeticiones
   requirió para sobrepasar ese límite.
```

Por ejemplo, si $c = 0.3 + 0.3i$, entonces:

```
   1ra iter: z =  0.30 + 0.30i  |z| = 0.42
   2da iter: z =  0.30 + 0.48i  |z| = 0.57
   3ra iter: z =  0.16 + 0.59i  |z| = 0.61
   4ta iter: z = -0.02 + 0.49i  |z| = 0.49
```

En este caso, z permanecerá limitado, aun hasta después de miles de iteraciones. Sin embargo, para $c = 0.5 + 1.0i$:

```
   1ra iter: z =  0.50 + 1.00i  |z| = 1.10
   2da iter: z = -0.25 + 2.00i  |z| = 2.00
   3ra iter: z = -3.44 + 0.00i  |z| = 3.40
   4ta iter: z = 12.32 + 1.00i  |z| = 12.4
   5ta iter: z = 151.1 +25.63i  |z| = 153.4
```

Veamos cómo el valor de *z*, en pocos pasos, explota a valores infinitos. En la iteración 10, el valor ya excederá la capacidad para que un computador pueda guardar el número en memoria. Sin embargo, podemos evitar hacer estos cálculos de *z* una vez su valor absoluto exceda 2.0, ya que se puede demostrar que una vez ese valor es alcanzado, *z* tiende a divergir.

El objetivo es llevar a cabo el cálculo para una serie de valores de *c*, y el número de iteraciones que se logra ejecutar antes de que $|z| > 2.0$ se grafique como función de la posición de *c* en el plano complejo (la parte real en el eje *x*, la parte imaginaria en el eje *y*). El *conjunto de Mandelbrot* es entonces el conjunto de números complejos *c* para los cuales el tamaño de $z^2 + c$ es finito, aún después de un número infinito de iteraciones. Una buena aproximación es, por ejemplo, realizar esta operación hasta un número grande de iteraciones (*1000* puede ser una buena opción), y asumir que si $|z| > 2$, el valor va a divergir.

Una función (definición) en *mandel_set.py* permite realizar esta operación para una grilla con un punto central *(x0,y0)* y un ancho de la grilla (*dx*).

mandel_set.py

```
def mandel1(x0=-0.21503361460851339,
            y0=0.67999116792639069,
            dx=0.2):

    import numpy as np

    # defina la grilla
    nx  = 300
    x   = np.linspace(x0-dx,x0+dx,nx)
    y   = np.linspace(y0-dx,y0+dx,nx)
    dat = np.zeros((nx,nx))

    # Calcular z con un loop, hasta 1000 veces
    for ix in range(nx):
```

```
        for iy in range(nx):
            cr = x[ix]
            ci = y[iy]
            c = complex(cr,ci)
            z = complex(0.0,0.0)
            for it in range(1000):
                z = c + z*z
                if (abs(z)>2.0):
                    break
            dat[ix,iy] = it+1
    dat = np.transpose(dat)
    dat = np.log10(dat)
    return dat
```

Primero, la función define la grilla, con 300 puntos para el eje real (x), y 300 para el eje imaginario (y), para un total de 90 000 puntos, para generar la matriz *dat*, donde se va a guardar el número de iteraciones necesarias para que $|z| > 2$.

```
# defina la grilla

nx   = 300
x    = np.linspace(x0-dx,x0+dx,nx)
y    = np.linspace(y0-dx,y0+dx,nx)
dat = np.zeros((nx,nx))
```

Después, para cada posición en la matriz *dat* (*for loops* de *ix* y *iy*), se define el valor de *c*:

```
cr = x[ix]
ci = y[iy]
c = complex(cr,ci)
```

Posteriormente, se ejecuta la iteración para calcular *z*:

```
z = complex(0.0,0.0)
for it in range(1000):
   z = c + z*z
      if (abs(z)>2.0):
         break
```

Cuando $|z| > 2$, el *loop* se detiene y la ubicación apropiada de la matriz *dat(x,y)* recibe el valor del número de iteraciones necesarias para cumplir esta condición.

```
dat[ix,iy] = it+1
```

La figura 9.2 muestra la matriz *dat* resultante, el *conjunto de Mandelbrot* en escala logarítmica con un punto central constante:

$$
\begin{aligned}
x_0 &= -0.21503361460851339 \\
y_0 &= 0.67999116792639069,
\end{aligned}
$$

con rangos de la grilla cada vez menores. En otras palabras, se muestran diferentes acercamientos (*zooms*) alrededor del mismo punto, en los que se muestran características de la estructura del conjunto de Mandelbrot autosimilares.

Un programa ejemplo que usa la función *mandel_set.py* para generar una figura (un solo panel de la figura 9.2) del conjunto de Mandelbrot se muestra a continuación:

plot_mandel.py

```
import numpy as np
import matplotlib.pyplot as plt

import mandel_set

x0 =-0.21503361460851339
```

```
y0 = 0.67999116792639069
dx = 0.4

fig = plt.figure(figsize=(10,10))
dat = mandel1(x0,y0,dx)
nx = np.shape(dat)[0]
ax = fig.add_subplot(3,3,i+1)
im = ax.imshow(dat,cmap='twilight_shifted',
        animated=True)
ax.set_title(f'Escala: {2*dx:.1e}')
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```

Problemas

1. Escriba un programa para evaluar la precisión en la definición de un número complejo en Python, usando la siguiente formula:

   $$e^z = e^{(x+yj)} = e^x * e^{yj} = (e^x cos(y)) + (e^x sin(y))j$$

   Para ello, se debe *programar* cada función para un número complejo z que indique el usuario. Asegúrese de que el programa dé como resultados las respuestas correctas para:

   ```
   exp( 1.1 + 2.3j) = -2.002 + 2.240j
   exp( 0.0 + 1.2j) =  0.362 + 0.932j
   exp(-0.5 + 0.0j) =  0.607 + 0.000j
   ```

   ¿Se encuentran diferencias entre la función predefinida *cmath.exp()* y los resultados de las otras fórmulas?

2. Utilizando el programa desarrollado en clase para el *conjunto de Mandelbrot*, genere tres figuras haciendo *zoom* en diferentes regiones.

Busque figuras que sean atractivas e interesantes. Note que el *conjunto de Mandelbrot* es autosimilar.

3. El artículo de *Scientific American* [9] muestra de una manera muy didáctica cómo se pueden generar los fractales; presenta el *conjunto de Julia* y su relación con el *conjunto de Mandelbrot*.

   En resumen, para el conjunto de Mandelbrot que se muestra en el código arriba, $z = 0.0 + 0.0j$ y c, determina el punto en el plano complejo (varía según la grilla que se quiere). ¿Qué pasaría si, en cambio, se define c como un valor fijo, y z juega el papel de punto inicial? Esto da como resultado el *conjunto de Julia*. Una figura distinta se espera dependiendo del valor de c que se elija.

   Genere tres figuras que crea que son atractivas del *conjunto de Julia*. Marque muy claramente los valores de c que utilizó, y los límites de su figura.

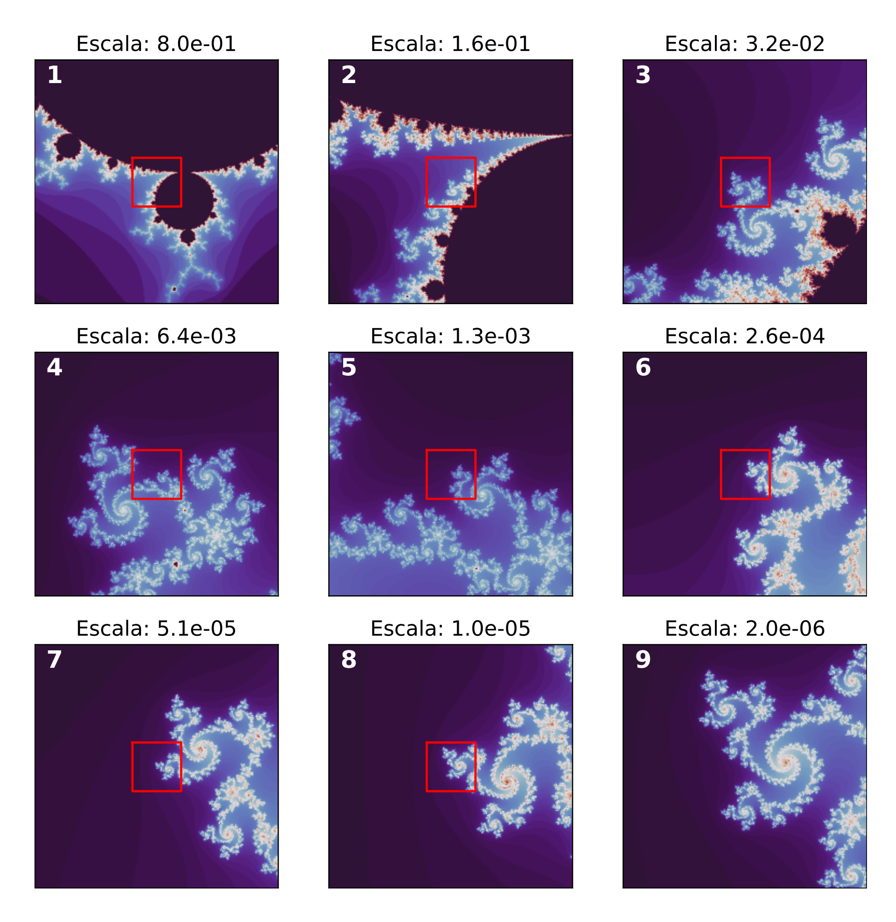


Figura 9.2. Conjunto de Mandelbrot con varios acercamientos (*zooms*) alrededor del punto $x = -0.21503361460851339$ y $y = 0.67999116792639069$, como se marca en cada panel. El recuadro rojo muestra la región de acercamiento del siguiente panel.

Bibliografía

[1] JM. Aiken, C. Aiken, and F. Cotton, *A python library for teaching computation to seismology students*, Seismological Research Letters **89** (2018), no. 3, 1165–1171.

[2] CC. Barton, R. Paul, and L. Pointe, *Fractals in the earth sciences*, Springer Science & Business Media, 1995.

[3] M. Beyreuther, R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann, *Obspy: A python toolbox for seismology*, Seismological Research Letters **81** (2010), no. 3, 530–533.

[4] WG. Blumberg, KT. Halbert, TA. Supinie, PT. Marsh, RL. Thompson, and JA. Hart, *Sharppy: An open-source sounding analysis toolkit for the atmospheric sciences*, Bulletin of the American Meteorological Society **98** (2017), no. 8, 1625–1636.

[5] E. Bonnet, O. Bour, NE. Odling, P. Davy, I. Main, P. Cowie, and B. Berkowitz, *Scaling of fracture systems in geological media*, Reviews of Geophysics **39** (2001), no. 3, 347–383.

[6] GA. Cox, WJ. Brown, L. Billingham, and R. Holme, *Magpysv: A python package for processing and denoising geomagnetic observatory data*, Geochemistry, Geophysics, Geosystems **19** (2018), no. 9, 3347–3363.

[7] FK. Dannemann Dugick, S. van der Lee, GA. Prieto, SN. Dybing, L. Toney, and HM. Cole, *Roses: Remote online sessions for emerging seismologists*, Seism. Res. Lett. **92** (2021), no. 4, 2657–2667.

[8] R. De Castro, *El universo LaTeX*, segunda ed., Unibiblos, Universidad Nacional de Colombia, Bogotá, Colombia, 2003.

[9] AK. Dewdney, *Beauty and profundity, the mandelbrot set and a flock of its cousins called julia*, Scientific American **257** (1987), no. 5, 118–122.

[10] G. Ekström, GA. Abers, and SC. Webb, *Determination of surface-wave phase velocities across usarray from noise and aki's spectral formulation*, Geophysical Research Letters **36** (2009), no. 18.

[11] VG. Gabriel, *Theory of complex numbers in structural geology*, Eos, Transactions American Geophysical Union **35** (1954), no. 2, 310–319.

[12] M. Goossens, F. Mittlebach, and A. Samarin, *The LaTeX Companion*, Addison-Wesley, 1994.

[13] JP. Grotzinger and DH. Rothman, *An abiotic model for stromatolite morphogenesis*, Nature **383** (1996), no. 6599, 423–425.

[14] CR. Harris, KJ. Millman, SJ. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, NJ. Smith, et al., *Array programming with numpy*, Nature **585** (2020), no. 7825, 357–362.

[15] NJ. Higham, *BibTeX: A Versatile Tool for LaTeX Users*, SIAM News **27** (1994), no. 1, 10–19.

[16] B. Jenny, B. Šavrič, ND. Arnold, BE. Marston, and CA. Preppernau, *A guide to selecting map projections for world and hemisphere maps*, Choosing a map projection, Springer, 2017, pp. 213–228.

[17] C. Jiang and MA. Denolle, *Noisepy: A new high-performance python tool for ambient-noise seismology*, Seismological Research Letters **91** (2020), no. 3, 1853–1866.

[18] J. Kam, PCD. Milly, and KA. Dunne, *Monthly time series of precipitation, air temperature, and net radiation for 2,673 gaged river basins worldwide: U.s. geological survey data release*, Tech. report, USGS, 2018.

[19] DE. Knuth, *The TeXbook*, Computers and Typesetting, Addison-Wesley, 1986.

[20] H. Kopka and PW. Daly, *A Guide to LaTeX2e. Document Preparation for Beginners and Advanced Users*, fourth ed., Addison-Wesley Publishing Company, 2004.

[21] L. Krischer, YA. Aiman, T. Bartholomaus, S. Donner, M. van Driel, K. Duru, K. Garina, K. Gessele, T. Gunawan, S. Hable, et al., *Seismo-live: An educational online library of jupyter notebooks for seismology*, Seismological Research Letters **89** (2018), no. 6, 2413–2419.

[22] L. Lamport, *LaTeX: A Document Preparation System*, second ed., Addison-Wesley Professional, 1994.

[23] B. Mandelbrot, *How long is the coast of britain? Statistical self-similarity and fractional dimension*, Science **156** (1967), no. 3775, 636–638.

[24] Met Office, *Cartopy: a cartographic python library with a matplotlib interface*, Exeter, Devon, 2010-2015.

[25] P. Poli and GA. Prieto, *Global rupture parameters for deep and intermediate-depth earthquakes*, Journal of Geophysical Research: Solid Earth **121** (2016), no. 12, 8871–8887.

[26] GA. Prieto, *The multitaper spectrum analysis package in python*, Seismological Research Letters **93** (2022), no. 3, 1922–1929.

[27] GA. Prieto, JF. Lawrence, and GC. Beroza, *Anelastic earth structure from the coherency of the ambient seismic field*, Journal of Geophysical Research: Solid Earth **114** (2009), no. B7.

[28] GA. Prieto, RL. Parker, and FL. Vernon, *A fortran 90 library for multitaper spectrum analysis*, Computers & Geosciences **35** (2009), no. 8, 1701–1710.

[29] GA. Prieto, PM. Shearer, FL. Vernon, and D. Kilb, *Earthquake source scaling and self-similarity estimation from stacking p and s spectra*, Journal of Geophysical Research: Solid Earth **109** (2004), no. B8.

[30] GA. Prieto, DJ. Thomson, FL. Vernon, PM. Shearer, and RL. Parker, *Confidence intervals for earthquake source parameters*, Geophysical Journal International **168** (2007), no. 3, 1227–1234.

[31] BR. Röbke and A. Vött, *The tsunami phenomenon*, Progress in Oceanography **159** (2017), 296–322.

[32] CH. Scholz and BB. Mandelbrot, *Fractals in geophysics*, Springer, 1989.

[33] PM. Shearer, *Introduction to seismology*, Cambridge University Press, 2019.

[34] H. Shen, *Interactive notebooks: Sharing the code*, Nature News **515** (2014), no. 7525, 151.

[35] JP. Snyder, *Map projections used by the us geological survey*, Tech. report, US Government Printing Office, 1982.

[36] DJ. Thomson, *Spectrum estimation and harmonic analysis*, Proceedings of the IEEE **70** (1982), no. 9, 1055–1096.

[37] JW. Tukey, *Conclusions vs decisions*, Technometrics **2** (1960), no. 4, 423–433.

[38] DL. Turcotte, *Fractals in geology and geophysics*, Pure and applied Geophysics **131** (1989), no. 1, 171–196.

[39] ———, *Chaos, fractals, nonlinear phenomena in earth sciences*, Reviews of Geophysics **33** (1995), no. S1, 341–343.

[40] L. Uieda, D. Tian, WJ. Leong, W. Schlitzer, L. Toney, M. Grund, M. Jones, J. Yao, K. Materna, T. Newton, A. Anant, M. Ziebarth, and P. Wessel, *PyGMT: A Python Interface for the Generic Mapping Tools*, Zenodo (2021).

[41] L. Uieda and P. Wessel, *Pygmt: Accessing the generic mapping tools from python*, AGU Fall Meeting Abstracts, vol. 2019, 2019, pp. NS21B–0813.

[42] P. Wessel, JF. Luis, L. Uieda, R. Scharroo, F. Wobbe, WHF. Smith, and D. Tian, *The generic mapping tools version 6*, Geochemistry, Geophysics, Geosystems **20** (2019), no. 11, 5556–5564.

[43] MJ. Williams, L. Schoneveld, Y. Mao, J. Klump, J. Gosses, H. Dalton, A. Bath, and S. Barnes, *Pyrolite: Python for geochemistry*, Journal of Open Source Software **5** (2020), no. 50, 2314.

Índice

Introducción a Python para geociencias
fue editado por el Centro Editorial de la Facultad
de Ciencias, Universidad Nacional de Colombia,
sede Bogotá. Se utilizaron como fuentes principales
Baskerllive y Fira Sans. Bogotá D. C.